

Visual C# 2.0

Lucian Sasu

May 22, 2006

Cuprins

1	Platforma Microsoft .NET	9
1.1	Prezentare generală	9
1.2	Arhitectura platformei Microsoft .NET	11
1.3	Componente ale lui .NET Framework	12
1.3.1	Microsoft Intermediate Language	12
1.3.2	Common Language Specification	12
1.3.3	Common Language Runtime	13
1.3.4	Common Type System	14
1.3.5	Metadata	15
1.3.6	Assemblies	16
1.3.7	Assembly cache	17
1.3.8	Garbage collection	17
1.4	Trăsături ale platformei .NET	17
2	Tipuri predefinite, tablouri, string-uri	21
2.1	Vedere generală asupra limbajului C#	21
2.2	Tipuri de date	23
2.2.1	Tipuri predefinite	24
2.2.2	Tipuri valoare	25
2.2.3	Tipul enumerare	29
2.3	Tablouri	36
2.3.1	Tablouri unidimensionale	36
2.3.2	Tablouri multidimensionale	37
2.4	Șiruri de caractere	41
2.4.1	Expresii regulate	43
3	Clase, instrucțiuni, spații de nume	45
3.1	Clase – vedere generală	45
3.2	Transmiterea de parametri	49
3.3	Conversii	55
3.3.1	Conversii implicite	55

3.3.2	Conversiile implicite ale expresiilor constante	57
3.3.3	Conversii explicite	57
3.3.4	Boxing și unboxing	60
3.3.5	Declarații de variabile și constante	62
3.3.6	Declarații de etichete	62
3.3.7	Instrucțiuni de selecție	63
3.3.8	Instrucțiuni de ciclare	64
3.3.9	Instrucțiuni de salt	66
3.3.10	Instrucțiunile try, throw, catch, finally	66
3.3.11	Instrucțiunile checked și unchecked	67
3.3.12	Instrucțiunea lock	67
3.3.13	Instrucțiunea using	67
3.4	Spații de nume	68
3.4.1	Declarații de spații de nume	69
3.4.2	Directiva using	70
4	Clase	75
4.1	Declararea unei clase	75
4.2	Membrii unei clase	76
4.3	Câmpuri	77
4.3.1	Câmpuri instanțe	77
4.3.2	Câmpuri statice	77
4.3.3	Câmpuri readonly	78
4.3.4	Câmpuri volatile	78
4.3.5	Inițializarea câmpurilor	79
4.4	Constante	79
4.5	Metode	80
4.5.1	Metode statice și nestatice	81
4.5.2	Metode externe	81
4.6	Proprietăți	81
4.7	Indexatori	87
4.8	Operatori	93
4.8.1	Operatori unari	93
4.8.2	Operatori binari	95
4.8.3	Operatori de conversie	96
4.8.4	Exemplu: clasa Fraction	97
4.9	Constructorii de instanță	99
4.10	Constructor static	100
4.11	Clase interioare	101
4.12	Destructorii	104

5	Clase (2). Moștenire. Interfețe. Structuri	107
5.1	Clase statice	107
5.2	Specializarea și generalizarea	109
5.2.1	Specificarea moștenirii	109
5.2.2	Apelul constructorilor din clasa de bază	110
5.2.3	Operatorii <i>is</i> și <i>as</i>	111
5.3	Clase <i>sealed</i>	112
5.4	Polimorfismul	112
5.4.1	Polimorfismul parametric	112
5.4.2	Polimorfismul ad-hoc	112
5.4.3	Polimorfismul de moștenire	113
5.4.4	<i>Virtual</i> și <i>override</i>	114
5.4.5	Modificatorul <i>new</i> pentru metode	115
5.4.6	Metode <i>sealed</i>	118
5.4.7	Exemplu folosind <i>virtual</i> , <i>new</i> , <i>override</i> , <i>sealed</i>	119
5.5	Clase și metode abstracte	121
5.6	Interfețe	122
5.6.1	Clase abstracte sau interfete?	128
5.7	Tipuri parțiale	128
5.8	Structuri	130
5.8.1	Structuri sau clase?	134
6	Delegați. Evenimente. Tratarea excepțiilor	137
6.1	Tipul delegat	137
6.1.1	Utilizarea delegaților pentru a specifica metode la run-time	138
6.1.2	Delegați statici	141
6.1.3	Multicasting	142
6.2	Evenimente	145
6.2.1	Publicarea și subscrierea	145
6.2.2	Evenimente și delegați	145
6.2.3	Comentarii	152
6.3	Tratarea excepțiilor	152
6.3.1	Tipul <i>Exception</i>	153
6.3.2	Aruncarea și prinderea excepțiilor	153
6.3.3	Reîncercarea codului	164
6.3.4	Compararea tehnicilor de manipulare a erorilor	166
6.3.5	Sugestie pentru lucrul cu excepțiile	168

7	ADO.NET	169
7.1	Ce reprezintă ADO.NET?	169
7.2	Furnizori de date în ADO.NET	170
7.3	Componentele unui furnizor de date	170
7.3.1	Clasele <i>Connection</i>	171
7.3.2	Clasele <i>Command</i>	172
7.3.3	Clasele <i>DataReader</i>	172
7.3.4	Clasele <i>DataAdapter</i>	172
7.3.5	Clasa <i>DataSet</i>	172
7.4	Obiecte <i>Connection</i>	172
7.4.1	Proprietăți	173
7.4.2	Metode	175
7.4.3	Evenimente	175
7.4.4	Stocarea stringului de conexiune în fișier de configurare	175
7.4.5	Gruparea conexiunilor	177
7.4.6	Mod de lucru cu conexiunile	177
7.5	Obiecte <i>Command</i>	178
7.5.1	Proprietăți	179
7.5.2	Metode	179
7.5.3	Utilizarea unei comenzi cu o procedură stocată	182
7.5.4	Folosirea comenzilor parametrizate	183
7.6	Obiecte <i>DataReader</i>	184
7.6.1	Proprietăți	185
7.6.2	Metode	185
7.6.3	Crearea și utilizarea unui <i>DataReader</i>	186
7.6.4	Utilizarea de seturi de date multiple	187
7.6.5	Accesarea datelor într-o manieră sigură din punct de vedere a tipului	187
8	ADO.NET (2)	189
8.1	Obiecte <i>DataAdapter</i>	189
8.1.1	Metode	190
8.1.2	Proprietăți	191
8.2	Clasa <i>DataSet</i>	191
8.2.1	Conținut	192
8.2.2	Clasa <i>DataTable</i>	192
8.2.3	Relații între tabele	195
8.2.4	Popularea unui <i>DataSet</i>	195
8.2.5	Clasa <i>DataTableReader</i>	196
8.2.6	Propagarea modificărilor către baza de date	197
8.3	Tranzacții în ADO.NET	199

8.4	Lucrul generic cu furnizori de date	201
8.5	Tipuri nulaabile	203
9	Colecții. Clase generice	205
9.1	Colecții	205
9.1.1	Iteratori pentru colecții	208
9.1.2	colecții de tip listă	209
9.1.3	Colecții de tip dicționar	210
9.2	Crearea unei colecții	211
9.2.1	Colecție iterabilă (stil vechi)	211
9.2.2	Colecție iterabilă (stil nou)	214
9.3	Clase generice	219
9.3.1	Metode generice	219
9.3.2	Tipuri generice	220
9.3.3	Constrângeri asupra parametrilor de genericitate	221
9.3.4	Interfețe și delegați generici	222
9.4	Colecții generice	223
9.4.1	Probleme cu colecțiile de obiecte	223
9.4.2	Colecții generice	223
10	ASP.NET	225
10.1	Anatomia unei pagini ASP.NET	226
10.1.1	Adăugarea unui control Web	226
10.1.2	Adăugarea scriptului <i>inline</i>	228
10.1.3	Code-behind	229
10.2	Clasa <i>Page</i>	230
10.2.1	Evenimentul <i>Init</i>	230
10.2.2	Evenimentul <i>Load</i>	231
10.2.3	Evenimentul <i>Unload</i>	231
10.3	Crearea unei pagini ASP.NET folosind Visual Studio.NET	231
10.4	Controale server	233
10.4.1	Postback	233
10.4.2	Data Binding	235
10.5	Controale Web	235
10.5.1	Label	236
10.5.2	Button	236
10.5.3	LinkButton	237
10.5.4	TextBox	237
10.5.5	CheckBox	237
10.5.6	RadioButton	237
10.5.7	DropDownList	237

11 ASP.NET (2)	239
11.1 Controale de validare	239
11.1.1 RequiredFieldValidator	240
11.1.2 RegularExpressionValidator	240
11.1.3 RangeValidator	240
11.1.4 CompareValidator	240
11.1.5 CustomValidator	241
11.1.6 ValidationSummary	242
11.2 Comunicarea cu browserul	242
11.2.1 Generarea programatică conținutului	243
11.2.2 Redirectarea	243
11.3 Cookies	244
11.4 Fișierul de configurare al aplicației Web	245
11.5 Fișierul <i>global.asax</i>	245
11.6 Managementul sesiunii	246
11.7 ViewState	247
11.8 Application	247
11.9 Alte puncte de interes în aplicații ASP.NET	247

Curs 1

Platforma Microsoft .NET

1.1 Prezentare generală

Platforma .NET 2.0 este un cadru de dezvoltare a softului, sub care se vor realiza, distribui și rula aplicațiile de tip forme Windows, aplicații WEB și servicii WEB. Ea constă în trei părți principale: Common Language Runtime, clasele specifice platformei și ASP.NET. O infrastructură ajutătoare, Microsoft .NET Compact Framework este un set de interfețe de programare care permite dezvoltatorilor realizarea de aplicații pentru dispozitive mobile precum telefoane inteligente și PDA-uri¹.

.NET Framework constituie un nivel de abstractizare între aplicație și nucleul sistemului de operare (sau alte programe), pentru a asigura portabilitatea codului; de asemenea integrează tehnologii care au fost lansate de către Microsoft începând cu mijlocul anilor 90 (COM, DCOM, ActiveX, etc) sau tehnologii actuale (servicii Web, XML).

Platforma constă câteva grupe de produse:

1. *Unelte de dezvoltare* - un set de limbaje (C#, Visual Basic .NET, J#, Managed C++, JScript.NET, Objective-C, Python, Smalltalk, Eiffel, Perl, Fortran, Cobol, Lisp, Haskell, Pascal, RPG, etc), un set de medii de dezvoltare (Visual Studio .NET, Visio), infrastructura .NET Framework, o bibliotecă cuprinzătoare de clase pentru crearea serviciilor Web (Web Services)², aplicațiilor Web (Web Forms) și aplicațiilor Windows (Windows Forms).
2. *Servere specializate* - un set de servere Enterprise .NET: SQL Server 2005, Exchange 2005, BizTalk Server 2006, etc, care pun la dispoziție

¹Definiția oficială Microsoft, martie 2005, www.microsoft.com/net/basics/glossary.asp

²Serviciilor Web - aplicații care oferă servicii folosind Web-ul ca modalitate de acces.

funcționalități diverse pentru stocarea bazelor de date, email, aplicații B2B³.

3. *Servicii Web* - cel mai notabil exemplu este .NET Passport - un mod prin care utilizatorii se pot autentifica pe site-urile Web vizitate, folosind un singur nume și o parolă pentru toate. Deși nu este omniprezent, multe site-uri îl folosesc pentru a ușura accesul utilizatorilor. Alt exemplu este MapPoint care permite vizualizarea, editarea și integrarea hartilor.
4. *Dispozitive* - noi dispozitive non-PC, programabile prin .NET Compact Framework, o versiune redusă a lui .NET Framework: Pocket PC Phone Edition, Smartphone, Tablet PC, Smart Display, XBox, set-top boxes, etc.

Motivul pentru care Microsoft a trecut la dezvoltarea acestei platforme este maturizarea industriei software, accentuându-se următoarele direcții:

1. *Aplicațiile distribuite* - sunt din ce în ce mai numeroase aplicațiile de tip client / server sau cele pe mai multe nivele (n -tier). Tehnologiile distribuite actuale cer de multe ori o mare afinitate față de producător și prezintă o carență acută a interoperării cu Web-ul. Viziunea actuală se depărtează de cea de tip client/server către una în care calculatoare, dispozitive inteligente și servicii conlucrează pentru atingerea scopurilor propuse. Toate acestea se fac deja folosind standarde Internet neproprietare (HTTP, XML, SOAP).
2. *Dezvoltarea orientată pe componente* - este de mult timp cerută simplificarea integrării componentelor software dezvoltate de diferiți producători. COM (Component Object Model) a realizat acest deziderat, dar dezvoltarea și distribuirea aplicațiilor COM este prea complexă. Microsoft .NET pune la dispoziție un mod mai simplu de a dezvolta și a distribui componente.
3. *Modificări ale paradigmei Web* - de-a lungul timpului s-au adus îmbunătățiri tehnologiilor Web pentru a simplifica dezvoltarea aplicațiilor. În ultimii ani, dezvoltarea aplicațiilor Web s-a mutat de la prezentare (HTML și adiacente) către capacitate sporită de programare (XML și SOAP).
4. *Alți factori de maturizare a industriei software* - reprezintă conștientizarea cererilor de interoperabilitate, scalabilitate, disponibilitate; unul din dezideratele .NET este de a oferi toate acestea.

³Bussiness to Bussiness - Comerț electronic între parteneri de afaceri, diferită de comerțul electronic între client și afacere (Bussiness to Consumer – B2C).

1.2 Arhitectura platformei Microsoft .NET

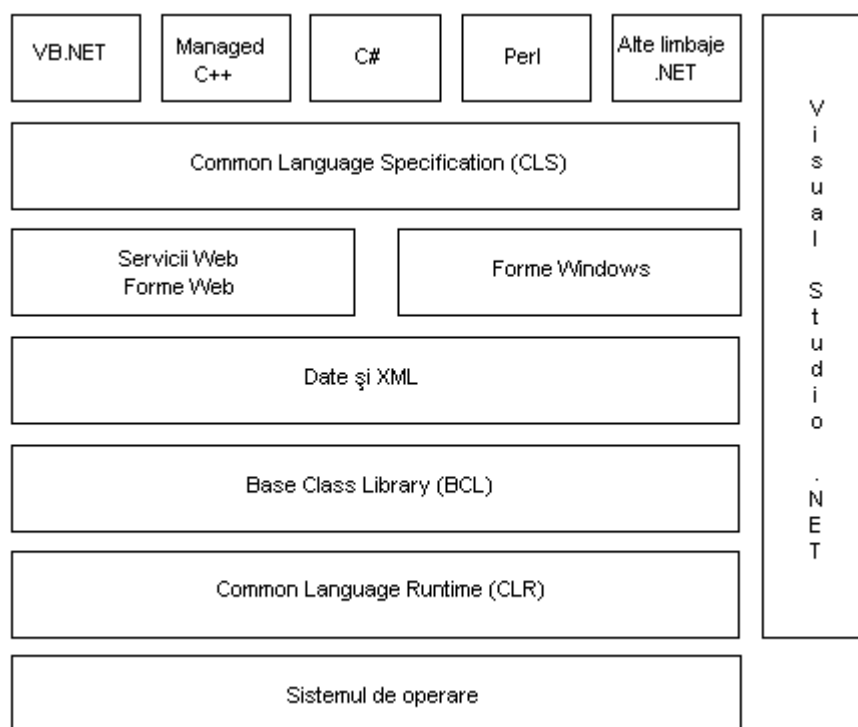


Figura 1.1: Arhitectura .NET

Figura 1.1 schematizează arhitectura platformei Microsoft .NET. Orice program scris într-unul din limbajele .NET este compilat în Microsoft Intermediate Language (MSIL), în concordanță cu Common Language Specification (CLS). Aceste limbaje sunt sprijinite de o bogată colecție de biblioteci de clase, ce pun la dispoziție facilități pentru dezvoltarea de Web Forms, Windows Forms și Web Services. Comunicarea dintre aplicații și servicii se face pe baza unor clase de manipulare XML și a datelor, ceea ce sprijină dezvoltarea aplicațiilor cu arhitectură *n-tier*. Base Class Library există pentru a asigura funcționalitate de nivel scăzut, precum operații de I/O, fire de execuție, lucrul cu șiruri de caractere, comunicație prin rețea, etc. Aceste clase sunt reunite sub numele de .NET Framework Class Library, ce permite dezvoltarea rapidă a aplicațiilor. La baza tuturor se află cea mai importantă componentă a lui .NET Framework - Common Language Runtime, care răspunde de execuția fiecărui program. Mediile de dezvoltare Visual Studio .NET nu sunt absolut necesare pentru a dezvolta aplicații, dar datorită uneltelor integrate de care dispune sunt cele mai bine adaptate pentru

crearea aplicațiilor. Evident, nivelul inferior este rezervat sistemului de operare. Trebuie spus că platforma .NET nu este exclusiv dezvoltată pentru sistemul de operare Microsoft Windows, ci și pentru arome de Unix (FreeBSD sau Linux - a se vedea proiectul Mono⁴).

1.3 Componente ale lui .NET Framework

1.3.1 Microsoft Intermediate Language

Una din uneltele de care dispune ingineria software este *abstractizarea*. Deseori vrem să ferim utilizatorul de detalii, să punem la dispoziția altora o interfață simplă, care să permită atingerea scopului, fără a fi necesare cunoașterea tuturor detaliilor. Dacă interfața rămâne neschimbată, se pot modifica toate detaliile interne, fără a afecta acțiunile celorlați beneficiari ai codului.

În cazul limbajelor de programare, s-a ajuns treptat la crearea unor nivele de abstractizare a codului rezultat la compilare, precum *p-code* (cel produs de compilatorul Pascal-P) și *bytecode* (binecunoscut celor care au lucrat în Java). Bytecode-ul Java, generat prin compilarea unui fișier sursă, este cod scris într-un limbaj intermediar care suportă POO. Bytecod-ul este în același timp o abstractizare care permite executarea codului Java, indiferent de platforma țintă, atâta timp cât această platformă are implementată o mașină virtuală Java, capabilă să “traducă” mai departe fișierul class în cod nativ.

Microsoft a realizat și el propria sa abstractizare de limbaj, aceasta numindu-se Common Intermediate Language. Deși există mai multe limbaje de programare de nivel înalt (C#, Managed C++, Visual Basic .NET, etc), la compilare toate vor produce cod în același limbaj intermediar: Microsoft Intermediate Language (MSIL, sau IL pe scurt). Asemănător cu bytecod-ul, IL are trăsături OO, precum abstractizarea datelor, moștenirea, polimorfismul, sau concepte care s-au dovedit a fi extrem de necesare, precum excepțiile sau evenimentele. De remarcat că această abstractizare de limbaj permite rularea aplicațiilor independent de platformă (cu aceeași condiție ca la Java: să existe o mașină virtuală pentru acea platformă).

1.3.2 Common Language Specification

Unul din scopurile .NET este de a sprijini integrarea limbajelor astfel încât programele, deși scrise în diferite limbaje, pot interopera, folosind din plin moștenirea, polimorfismul, încapsularea, excepțiile, etc. Dar limbajele nu

⁴www.go-mono.com

sunt identice: unele suportă supraîncărcarea operatorilor (Managed C++, C#), altele nu (Visual Basic .NET); unele sunt case sensitive, altele nu. Pentru a se asigura totuși interoperabilitatea codului scris în diferite limbaje, Microsoft a publicat Common Language Specification (CLS), un subset al lui CTS (Common Type System, vezi 1.3.4), conținând specificații de reguli necesare pentru integrarea limbajelor.

CLS definește un set de reguli pentru compilatoarele .NET, asigurând faptul că fiecare compilator va genera cod care interferează cu platforma (mai exact, cu CLR-ul — vezi mai jos) într-un mod independent de limbajul sursă. Obiectele și tipurile create în diferite limbaje pot interacționa fără probleme suplimentare. Combinația CTS/CLS realizează de fapt interoperarea limbajelor. Concret, se poate ca o clasă scrisă în C# să fie moștenită de o clasă scrisă în Visual Basic care aruncă excepții ce sunt prinse de cod scris în C++ sau J#.

1.3.3 Common Language Runtime

CLR este de departe cea mai importantă parte componentă a lui .NET Framework. Este responsabilă cu managementul și execuția codului scris în limbaje .NET, aflat în format IL; este foarte similar cu Java Virtual Machine. CLR instanțiază obiectele, face verificări de securitate, depune obiectele în memorie, disponibilizează memoria prin garbage collection.

În urma compilării unei aplicații rezultă un fișier cu extensia exe, dar care nu este un executabil portabil Windows, ci un executabil portabil .NET (.NET PE). Acest cod nu este deci un executabil nativ, ci se va rula de către CLR, întocmai cum un fișier class este rulat în cadrul JVM. CLR folosește tehnologia compilării JIT - o implementare de mașină virtuală, în care o metodă sau o funcție, în momentul în care este apelată pentru prima oară, este tradusă în cod mașină. Codul translatat este depus într-un cache, evitând-se astfel recompilarea ulterioară. Există 3 tipuri de compilatoare JIT:

1. *Normal JIT* - a se vedea descrierea de mai sus.
2. *Pre-JIT* - compilează întregul cod în cod nativ singură dată. În mod normal este folosit la instalări.
3. *Econo-JIT* - se folosește pe dispozitive cu resurse limitate. Compilează codul IL bit cu bit, eliberând resursele folosite de codul nativ ce este stocat în cache.

În esență, activitatea unui compilator JIT este destinată a îmbunătăți performanța execuției, ca alternativă la compilarea repetată a aceleiași bucăți de cod în cazul unor apelări multiple. Unul din avantajele mecanismului JIT apare în clipa în care codul, o dată ce a fost compilat, se execută pe diverse procesoare; dacă mașina virtuală este bine adaptată la noua platformă, atunci acest cod va beneficia de toate optimizările posibile, fără a mai fi nevoie recompilarea lui (precum în C++, de exemplu).

1.3.4 Common Type System

Pentru a asigura interoperabilitatea limbajelor din .NET Framework, o clasă scrisă în C# trebuie să fie echivalentă cu una scrisă în VB.NET, o interfață scrisă în Managed C++ trebuie să fie perfect utilizabilă în Managed Cobol. Toate limbajele care fac parte din pleiada .NET trebuie să aibe un set comun de concepte pentru a putea fi integrate. Modul în care acest deziderat s-a transformat în realitate se numește Common Type System (CTS); orice limbaj trebuie să recunoască și să poată manipula niște tipuri comune.

O scurtă descriere a unor facilități comune (ce vor fi exhaustiv enumerate și tratate în cadrul prezentării limbajului C#):

1. *Tipuri valoare* - în general, CLR-ul (care se ocupă de managementul și execuția codului IL, vezi mai jos) suportă două tipuri diferite: tipuri valoare și tipuri referință. Tipurile valoare reprezintă tipuri alocate pe stivă și nu pot avea valoare de null. Tipurile valoare includ tipurile primitive, structuri și enumerări. Datorită faptului că de regulă au dimensiuni mici și sunt alocate pe stivă, se manipulează eficient, reducând overhead-ul cerut de mecanismul de garbage collection.
2. *Tipuri referință* - se folosesc dacă variabilele de un anumit tip cer resurse de memorie semnificative. Variabilele de tip referință conțin adrese de memorie heap și pot fi null. Transferul parametrilor se face rapid, dar referințele induc un cost suplimentar datorită mecanismului de garbage collection.
3. *Boxing și unboxing* - motivul pentru care există tipuri primitive este același ca și în Java: performanța. Însă orice variabilă în .NET este compatibilă cu clasa Object, rădăcina ierarhiei existente în .NET. De exemplu, *int* este un alias pentru *System.Int32*, care se derivează din *System.ValueType*. Tipurile valoare se stochează pe stivă, dar pot fi oricând convertite într-un tip referință memorat în heap; acest mecanism se numește *boxing*. De exemplu:

```
int i = 1; //i - un tip valoare  
object box = i; //box - un obiect referinta
```

Când se face boxing, se obține un obiect care poate fi gestionat la fel ca oricare altul, făcându-se abstracție de originea lui.

Inversa boxing-ului este unboxing-ul, prin care se poate converti un obiect în tipul valoare echivalent, ca mai jos:

```
int j = (int)box;
```

unde operatorul de conversie este suficient pentru a converti de la un obiect la o variabilă de tip valoare.

4. *Clase, proprietăți, indexatori* - platforma .NET suportă pe deplin programarea orientată pe obiecte, concepte legate de obiecte (încapsularea, moștenirea, polimorfismul) sau trăsături legate de clase (metode, câmpuri, membri statici, vizibilitate, accesibilitate, tipuri interioare, etc). De asemenea se includ trăsături precum proprietăți, indexatori, evenimente.
5. *Interfețe* - reprezintă același concept precum clasele abstracte din C++ (dar conținând doar funcții virtuale pure), sau interfețele Java. O clasă care se derivează dintr-o interfață trebuie să implementeze toate metodele acelei interfețe. Se permite implementarea simultană a mai multor interfețe (în rest moștenirea claselor este simplă).
6. *Delegați* - inspirați de pointerii la funcții din C, ce permit programarea generică. Reprezintă versiunea “sigură” a pointerilor către funcții din C/C++ și sunt mecanismul prin care se tratează evenimentele.

Există numeroase componente ale lui CLR care îl fac cea mai importantă parte a lui .NET Framework: metadata, assemblies, assembly cache, reflection, garbage collection.

1.3.5 Metadata

Metadata înseamnă în general “date despre date”. În cazul .NET, ea reprezintă informații destinate a fi citite de către mașină, nu de utilizatorul uman. Este stocată împreună cu codul pe care îl descrie. Pe baza metadatai, CLR știe cum să instanțieze obiectele, cum să le apeleze metodele, cum să acceseze proprietățile. Printr-un mecanism numit reflection, o aplicație (nu

neapărat CLR) poate să interogheze această metadată și să afle ce expune un obiect.

Mai pe larg, metadata conține o declarație a fiecărui tip și câte o declarație pentru fiecare metodă, câmp, proprietate, eveniment al tipului respectiv. Pentru fiecare metodă implementată, metadata conține informație care permite încărcătorului clasei respective să localizeze corpul metodei. De asemenea mai poate conține declarații despre *cultura* aplicației respective, adică despre localizarea ei (limba folosită în partea de interfață utilizator). Mecanismul prin care aceste informații se folosesc pe parcursul rulării de către aplicații se numește *reflection*.

1.3.6 Assemblies

Un assembly reprezintă un bloc funcțional al unei aplicații .NET. El formează *unitatea fundamentală de distribuire, versionare, reutilizare și permisiuni de securitate*. La runtime, un tip de date există în interiorul unui assembly (și nu poate exista în exteriorul acestuia). Un assembly conține metadata care sunt folosite de către CLR. Scopul acestor “assemblies” este să se asigure dezvoltarea softului în mod “plug-and-play”. Dar metadatale nu sunt suficiente pentru acest lucru; mai sunt necesare și “manifestele”.

Un manifest reprezintă metadata despre assembly-ul care găzduiește tipurile de date. Conține numele assembly-ului, informația despre versiune, referiri la alte assemblies, o listă a tipurilor în assembly, permisiuni de securitate și altele.

Un assembly care este împărțit între mai multe aplicații are de asemenea un *shared name*. Această informație care este unică este opțională, neapărând în manifestul unui assembly dacă acesta nu a fost gândit ca o aplicație partajată.

Deoarece un assembly conține date care îl descriu, instalarea lui poate fi făcută copiind assemblyul în directorul destinație dorit. Când se dorește rularea unei aplicații conținute în assembly, manifestul va instrui mediul .NET despre modulele care sunt conținute în assembly. Sunt folosite de asemenea și referințele către orice assembly extern de care are nevoie aplicația.

Versionarea este un aspect deosebit de important pentru a se evita așa-numitul “DLL Hell”. Scenariile precedente erau de tipul: se instalează o aplicație care aduce niște fișiere .dll necesare pentru functionare. Ulterior, o altă aplicație care se instalează suprascrie aceste fișiere (sau măcar unul din ele) cu o versiune mai nouă, dar cu care vechea aplicație nu mai funcționează corespunzător. Reinstalarea vechii aplicații nu rezolvă problema, deoarece a doua aplicație nu va funcționa. Deși fișierele dll conțin informație relativ la versiune în interiorul lor, ea nu este folosită de către sistemul de operare,

ceea ce duce la probleme. O soluție la această dilemă ar fi instalarea fișierelor dll în directorul aplicației, dar în acest mod ar dispărea reutilizarea acestor biblioteci.

1.3.7 Assembly cache

Assembly cache este un director aflat în mod normal în directorul %windir%\Assemblies. Atunci când un assembly este instalat pe o mașină, el va fi adăugat în assembly cache. Dacă un assembly este descărcat de pe Internet, el va fi stocat în assembly cache, într-o zonă tranzientă. Aplicațiile instalate vor avea assemblies într-un assembly cache global.

În acest assembly cache vor exista versiuni multiple ale aceluiași assembly. Dacă programul de instalare este scris corect, va evita suprascrierea assembly-urilor deja existente (și care funcționează perfect cu aplicațiile instalate), adăugând doar noul assembly. Este un mod de rezolvare a problemei “DLL Hell”, unde suprascrierea unei biblioteci dinamice cu o variantă mai nouă putea duce la nefuncționarea corespunzătoare a aplicațiilor anterior instalate. CLR este cel care decide, pe baza informațiilor din manifest, care este versiunea corectă de assembly de care o aplicație are nevoie. Acest mecanism pune capăt unei epoci de tristă amintire pentru programatori și utilizatori.

1.3.8 Garbage collection

Managementul memoriei este una din sarcinile cele mai consumatoare de resurse umane. Garbage collection este mecanismul care se declanșează atunci când alocatorul de memorie răspunde negativ la o cerere de alocare de memorie. Implementarea este de tip “mark and sweep”: se presupune inițial că toată memoria alocată se poate disponibiliza, după care se determină care din obiecte sunt referite de variabilele aplicației; cele care nu mai sunt referite sunt dealocate, celelalte zone de memorie sunt compactate. Obiectele a căror dimensiune de memorie este mai mare decât un anumit prag nu mai sunt mutate, pentru a nu crește semnificativ penalizarea de performanță.

În general, CLR este cel care se ocupă de apelarea mecanismului de garbage collection. Totuși, la dorință, programatorul poate cere rularea lui.

1.4 Trăsături ale platformei .NET

Prin prisma celor prezentate până acum putem rezuma următoarele trăsături:

- **Dezvoltarea multilimbaj:** Deoarece există mai multe limbaje pentru această platformă, este mai ușor de implementat părți specifice în limbajele cele mai adecvate. Numarul limbajelor curent implementate este mai mare decât 10. Această dezvoltare are în vedere și debugging-ul aplicațiilor dezvoltate în mai multe limbaje.
- **Independența de procesor și de platformă:** IL este independent de procesor. O dată scrisă și compilată, orice aplicație .NET (al cărei management este făcut de către CLR) poate fi rulată pe orice platformă. Datorită CLR-ului, aplicația este izolată de particularitățile hardware sau ale sistemului de operare.
- **Managementul automat al memoriei:** Problemele de tipul memory leakage nu mai trebuie să preocupe programatorii; overhead-ul indus de către mecanismul de garbage collection este suportabil, fiind implementat în sisteme mult mai timpurii.
- **Suportul pentru versionare:** Ca o lecție învățată din perioada de “DLL Hell”, versionarea este acum un aspect de care se ține cont. Dacă o aplicație a fost dezvoltată și testată folosind anumite componente, instalarea unei componente de versiune mai nouă nu va atenta la buna funcționare a aplicației în discuție: cele două versiuni vor coexista pașnic, alegerea lor fiind făcută pe baza manifestelor.
- **Sprijinirea standardelor deschise:** Nu toate dispozitivele rulează sisteme de operare Microsoft sau folosesc procesoare Intel. Din această cauză orice este strâns legat de acestea este evitat. Se folosește XML și cel mai vizibil descendent al acestuia, SOAP. Deoarece SOAP este un protocol simplu, bazat pe text, foarte asemănător cu HTTP⁵, el poate trece ușor de firewall-uri, spre deosebire de DCOM sau CORBA.
- **Distribuirea ușoară:** Actualmente instalarea unei aplicații sub Windows înseamnă copierea unor fișiere în niște directoare anume, modificarea unor valori în regiștri, instalare de componente COM, etc. Dezinstalarea completă a unei aplicații este în majoritatea cazurilor o utopie. Aplicațiile .NET, datorită metadatelor și reflectării trec de aceste probleme. Se dorește ca instalarea unei aplicații să nu însemne mai mult decât copierea fișierelor necesare într-un director, iar dezinstalarea aplicației să se facă prin ștergerea acelui director.
- **Arhitectură distribuită:** Noua filosofie este de a asigura accesul la servicii Web distribuite; acestea conlucrează la obținerea informației

⁵Folosit la transmiterea paginilor Web pe Internet

dorite. Platforma .NET asigură suport și unelte pentru realizarea acestui tip de aplicații.

- **Interoperabilitate cu codul “unmanaged”:** Codul “unmanaged” se referă la cod care nu se află în totalitate sub controlul CLR. El este rulat de CLR, dar nu beneficiază de CTS sau garbage collection. Este vorba de apelurile funcțiilor din DLL-uri, folosirea componentelor COM, sau folosirea de către o componentă COM a unei componente .NET. Codul existent se poate folosi în continuare.
- **Securitate:** Aplicațiile bazate pe componente distribuite cer automat securitate. Modalitatea actuală de securizare, bazată pe drepturile contului utilizatorului, sau cel din Java, în care codul suspectat este rulat într-un “sandbox”, fără acces la resursele critice este înlocuit în .NET de un control mai fin, pe baza metadatelor din assembly (zona din care provine - ex. Internet, intranet, mașina locală, etc) precum și a politicilor de securitate ce se pot seta.

Curs 2

Vedere generală asupra limbajului C#. Tipuri predefinite. Tablouri. Șiruri de caractere

2.1 Vedere generală asupra limbajului C#

C#¹ este un limbaj de programare modern, simplu, obiect-orientat. Este foarte asemănător cu Java și C++, motiv pentru care curba de învățare este foarte lină.

Un prim exemplu de program, care conține o serie de elemente ce se vor întâlni în continuare, este:

```
using System;
class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine("Hello world!");
    }
}
```

Prima linie *using System;* este o directivă care specifică faptul că se vor folosi clasele care sunt incluse în spațiul de nume² *System*; un spațiu de nume este o colecție de tipuri sau o grupare de alte spații de nume care pot

¹ “#” se pronunță “sharp”

²Eng: namespace

fi folosite într-un program (detalii vor fi date mai târziu). În cazul de față, clasa care este folosită din acest spațiu de nume este *Console*. Mai departe, orice program este conținut într-o clasă, în cazul nostru *HelloWorld*. Punctul de intrare în aplicație este metoda *Main*, care nu preia în acest exemplu nici un argument din linia de comandă și nu returnează explicit un indicator de stare a terminării.

Pentru operațiile de intrare-ieșire cu consola, se folosește clasa *Console* din spațiul de nume *System*; pentru această clasă se apelează metoda statică *WriteLine*, care afișează pe ecran mesajul, după care face trecerea la linie nouă.

O variantă ușor modificată a programului de mai sus, în care se salută persoanele ale căror nume este transmis prin linia de comandă:

```
using System;
class HelloWorld
{
    public static void Main( String[] args)
    {
        for( int i=0; i<args.Length; i++)
        {
            Console.WriteLine( "Hello {0}", args[i]);
        }
    }
}
```

În exemplul de mai sus metoda principală preia o listă de parametri transmiși din linia de comandă (un șir de obiecte de tip *String*) și va afișa pentru fiecare nume "Hello" urmat de numele de indice *i* (numerotarea parametrilor începe de la 0). Construcția "{0}" va fi înlocuită cu primul argument care urmează după "Hello {0}". La executarea programului de mai sus în forma: `HelloWorld Ana Dan`, se va afișa pe ecran:

```
Hello Ana
Hello Dan
```

Metoda *Main* poate să returneze o valoare întreagă, care să fie folosită de către sistemul de operare pentru a semnala dacă procesul s-a încheiat cu succes sau nu³. Menționăm faptul că limbajul este case-sensitive⁴.

Ca metode de notare Microsoft recomandă folosirea următoarelor două convenții:

³De exemplu în fișiere de comenzi prin testarea variabilei de mediu `ERRORLEVEL`

⁴Face distincție între litere mari și mici

- Convenție Pascal, în care prima literă a fiecărui cuvânt se scrie ca literă mare
- convenția tip "cămilă" este la fel ca precedenta, dar primul caracter al primului cuvânt nu este scris ca literă mare.

În general, convenția tip Pascal este folosită pentru tot ce este vizibil din exteriorul unei clase, precum clase, metode, proprietăți, etc. Doar parametrii metodelor se scriu cu convenția cămilă. Se recomandă evitarea folosirii notației ungare (numele unei entități trebuie să se refere la semantica ei, nu la tipul de reprezentare).

Cuvintele cheie (cuvinte rezervate care nu pot fi folosite ca și identificatori) din C# sunt date în tabelul 2.1:

Tabelul 2.1: Cuvinte cheie în C#

abstract	base	bool	break	byte
case	catch	char	checked	class
const	continue	decimal	default	delegate
do	double	else	enum	event
explicit	extern	false	finally	fixed
float	for	foreach	goto	if
implicit	in	int	interface	internal
is	lock	long	namespace	new
null	object	operator	out	override
params	private	protected	public	readonly
ref	return	sbyte	sealed	short
sizeof	static	string	struct	switch
this	throw	true	try	typeof
uint	ulong	unchecked	unsafe	ushort
using	virtual	void	while	

2.2 Tipuri de date

C# prezintă două tipuri de date: *tipuri valoare* și *tipuri referință*. Tipurile valoare includ tipurile simple (ex. *char*, *int*, *float*)⁵, tipurile enumerare

⁵De fapt acestea sunt structuri, prezentate în alt curs

și structură și au ca principale caracteristici faptul că ele conțin direct datele referite și sunt alocate pe stivă sau *inline* într-un struct. Tipurile referință includ tipurile clasă, interfață, delegat și tablou, toate având proprietatea că variabilele de acest tip stochează referințe către obiecte. Demn de remarcat este că toate tipurile de date sunt derivate (direct sau nu) din tipul `System.Object`, punând astfel la dispoziție un mod unitar de tratare a lor.

2.2.1 Tipuri predefinite

C# conține un set de tipuri predefinite, pentru care nu este necesară includerea vreunui spațiu de nume via directiva `using`: `string`, `object`, tipurile întregi cu semn și fără semn, tipuri numerice în virgulă mobilă, tipurile `bool` și `decimal`.

Tipul `string` este folosit pentru manipularea șirurilor de caractere codificate Unicode; conținutul obiectelor de tip `string` nu se poate modifica⁶. Clasa `object` este rădăcina ierarhiei de clase din .NET, la care orice tip (inclusiv un tip valoare) poate fi convertit.

Tipul `bool` este folosit pentru a reprezenta valorile logice true și false. Tipul `char` este folosit pentru a reprezenta caractere Unicode, reprezentate pe 16 biți. Tipul `decimal` este folosit pentru calcule în care erorile determinate de reprezentarea în virgulă mobilă sunt inacceptabile, el punând la dispoziție 28 de cifre zecimale semnificative.

Tabelul de mai jos conține lista tipurilor predefinite, arătând totodată cum se scriu valorile corespunzătoare:

Tabelul 2.2: Tipuri predefinite.

Tip	Descriere	Exemplu
<code>object</code>	rădăcina oricărui tip	<code>object a = null;</code>
<code>string</code>	o secvență de caractere Unicode	<code>string s = "hello";</code>
<code>sbyte</code>	tip întreg cu semn, pe 8 biți	<code>sbyte val = 12;</code>
<code>short</code>	tip întreg cu semn, pe 16 biți	<code>short val = 12;</code>
<code>int</code>	tip întreg cu semn, pe 32 biți	<code>int val = 12;</code>
<code>long</code>	tip întreg cu semn, pe 64 biți	<code>long val1 = 12;</code> <code>long val2=34L;</code>
<code>byte</code>	tip întreg fără semn, pe 8 biți	<code>byte val = 12;</code>
<code>ushort</code>	tip întreg fără semn, pe 16 biți	<code>ushort val = 12;</code>
<code>uint</code>	tip întreg fără semn, pe 32 biți	<code>uint val = 12;</code>

⁶Spunem despre un `string` că este *invariabil* - engl. *immutable*

Tabelul 2.2 (continuare)

Tip	Descriere	Exemplu
ulong	tip întreg fără semn, pe 64 de biți	ulong val1=12; ulong val2=34U; ulong val3=56L; ulong va;4=76UL;
float	tip cu virgulă mobilă, simplă precizie	float val=1.23F;
double	tip în virgulă mobilă, dublă precizie	double val1=1.23; double val2=4.56D;
bool	tip boolean	bool val1=false; bool val2=true;
char	tip caracter din setul Unicode	char val='h';
decimal	tip zecimal cu 28 de cifre semnificative	decimal val=1.23M;

Fiecare din tipurile predefinite este un alias pentru un tip pus la dispoziție de sistem. De exemplu, string este alias pentru clasa System.String, int este alias pentru System.Int32.

2.2.2 Tipuri valoare

C# pune programatorului la dispoziție tipuri valoare, care sunt fie structuri, fie enumerări. Există un set predefinit de structuri numite *tipuri simple*, identificate prin cuvinte rezervate. Un tip simplu este fie de tip numeric⁷, fie boolean. Tipurile numerice sunt tipuri întregi, în virgulă mobilă sau decimal. Tipurile întregi sunt sbyte, byte, short, ushort, int, uint, long, ulong, char; cele în virgulă mobilă sunt float și double. Tipurile enumerare se pot defini de către utilizator.

Toate tipurile valoare derivează din clasa System.ValueType, care la rândul ei este derivată din clasa object (alias pentru System.Object). Nu este posibil ca dintr-un tip valoare să se deriveze. Atribuirea pentru un astfel de tip înseamnă copierea valorii (clonarea) dintr-o parte în alta.

Structuri

Un tip structură este un tip valoare care poate să conțină declarații de constante, câmpuri, metode, proprietăți, indexatori, operatori, constructori sau tipuri imbricate. Vor fi descrise în într-un capitol următor.

⁷Engl: integral type

Tipuri simple

C# pune are predefinit un set de tipuri structuri, numite tipuri simple. Tipurile simple sunt identificate prin cuvinte rezervate, dar acestea reprezintă doar alias-uri pentru tipurile struct corespunzătoare din spațiul de nume System; corespondența este dată în tabelul de mai jos:

Tabelul 2.3: Tipuri simple și corespondențele lor cu tipurile din spațiul de nume System.

Tabelul 2.3

Cuvânt rezervat	Tipul alias
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Deoarece un tip simplu este un alias pentru un tip struct, orice tip simplu are membri. De exemplu, tipul `int`, fiind un tip alias pentru `System.Int32`, următoarele declarații sunt legale:

```
int i = int.MaxValue;    //constanta System.Int32.MaxValue
string s = i.ToString(); //metoda System.Int32.ToString()
string t = 3.ToString(); //idem
//double d = Double.ParseDouble("3.14");
```

Tipuri întregi

C# suportă nouă tipuri întregi: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` și `char`. Acestea au următoarele dimensiuni și domeniu de valori:

- `sbyte` reprezintă tip cu semn pe 8 biți, cu valori de la -128 la 127;

- byte reprezintă tip fără semn pe 8 biți, între 0 și 255;
- short reprezintă tip cu semn pe 16 biți, între -32768 și 32767;
- ushort reprezintă tip fără semn pe 16 biți, între 0 și 65535;
- int reprezintă tip cu semn pe 32 de biți, între -2^{31} și $2^{31} - 1$;
- uint reprezintă tip fără semn pe 32 de biți, între 0 și $2^{32} - 1$;
- long reprezintă tip cu semn pe 64 de biți, între -2^{63} și $2^{63} - 1$;
- ulong reprezintă tip fără semn pe 64 de biți, între 0 și $2^{64} - 1$;
- char reprezintă tip fără semn pe 16 biți, cu valori între 0 și 65535. Mulțimea valorilor posibile pentru char corespunde setului de caractere Unicode.

Reprezentarea unei variabile de tip întreg se poate face sub formă de șir de cifre zecimale sau hexazecimale, urmate eventual de un prefix. Numerele exprimate în hexazecimal sunt prefixate cu “0x” sau “0X”. Regulile după care se asignează un tip pentru o valoare sunt:

1. dacă șirul de cifre nu are un sufix, atunci el este considerat ca fiind primul tip care poate să conțină valoarea dată: int, uint, long, ulong;
2. dacă șirul de cifre are sufixul u sau U, el este considerat ca fiind din primul tip care poate să conțină valoarea dată: uint, ulong;
3. dacă șirul de cifre are sufixul l sau L, el este considerat ca fiind din primul tip care poate să conțină valoarea dată: long, ulong;
4. dacă șirul de cifre are sufixul ul, uL, Ul, UL, lu, lU, Lu, LU, el este considerat ca fiind din tipul ulong.

Dacă o valoare este în afara domeniului lui ulong, apare o eroare la compilare.

Literalii de tip caracter au forma: ‘caracter’ unde “caracter” poate fi exprimat printr-un caracter, printr-o secvență escape simplă, secvență escape hexazecimală sau secvență escape Unicode. În prima formă poate fi folosit orice caracter exceptând apostrof, backslash și new line. Secvență escape simplă poate fi: `\'`, `\"`, `\\`, `\0`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, cu semnificațiile cunoscute din C++. O secvență escape hexazecimală începe cu `\x` urmat de 1–4 cifre hexa. Deși ca reprezentare, char este identic cu ushort, nu toate operațiile ce se pot efectua cu ushort sunt valabile și pentru char.

În cazul în care o operație aritmetică produce un rezultat care nu poate fi reprezentat în tipul destinație, comportamentul depinde de utilizarea operatorilor sau a declarațiilor *checked* și *unchecked* (care se pot utiliza în sursă sau din linia de compilare): în context *checked*, o eroare de depășire duce la aruncarea unei excepții de tip `System.OverflowException`. În context *unchecked*, eroarea de depășire este ignorată, iar biții semnificativi care nu mai încap în reprezentare sunt eliminați.

Exemplu:

```
byte i=255;
unchecked
{
    i++;
} // i va avea valoarea 0, nu se semnaleaza eroare
checked
{
    i=255;
    i++; // se va arunca exceptie System.OverflowException
}
```

Pentru expresiile aritmetice care conțin operatorii `++`, `--`, `+`, `-` (unar și binar), `*`, `/` și care nu sunt conținute în interiorul unui bloc de tip *checked*, comportamentul este specificat prin intermediul opțiunii `/checked[+|-]` dat din linia de comandă pentru compilator. Dacă nu se specifică nimic, atunci se va considera implicit *unchecked*.

Tipuri în virgulă mobilă

Sunt prezente 2 tipuri numerice în virgulă mobilă: `float` și `double`. Tipurile sunt reprezentate folosind precizie de 32, respectiv 64 de biți, folosind formatul IEC 60559, care permit reprezentarea valorilor de “0 pozitiv” și “0 negativ” (se comportă la fel, dar anumite operații duc la obținerea acestor două valori), $+\infty$ și $-\infty$ (obținute prin împărțirea unui număr strict pozitiv, respectiv strict negativ la 0), a valorii Not-a-Number (NaN) (obținută prin operații în virgulă mobilă invalide, de exemplu $0/0$ sau $\sqrt{-1}$), precum și un set finit de numere. Tipul `float` poate reprezenta valori cuprinse între 1.5×10^{-45} și 3.4×10^{38} (și din domeniul negativ corespunzător), cu o precizie de 7 cifre. `Double` poate reprezenta valori cuprinse între 5.0×10^{-324} și 1.7×10^{308} cu o precizie de 15-16 cifre.

Operațiile cu floating point nu duc niciodată la apariția de excepții, dar ele pot duce, în caz de operații invalide, la valori 0, infinit sau NaN.

Literalii care specifică un număr reprezentat în virgulă mobilă au forma: literal-real::

cifre-zecimale . cifre-zecimale exponent_{optional} sufix-de-tip-real_{optional}
 . cifre-zecimale exponent_{optional} sufix-de-tip-real_{optional}
 cifre-zecimale exponent sufix-de-tip-real_{optional}
 cifre-zecimale sufix-de-tip-real,

unde

exponent::

e semn_{optional} cifre-zecimale

E semn_{optional} cifre-zecimale,

semn este + sau -, sufix-de-tip-real este F, f, D, d. Dacă nici un sufix de tip real nu este specificat, atunci literalul dat este de tip double. Sufixul f sau F specifică tip float, d sau D specifică double. Dacă literalul specificat nu poate fi reprezentat în tipul precizat, apare eroare de compilare.

Tipul decimal

Este un tip de date reprezentat pe 128 de biți, gândit a fi folosit în calcule financiare sau care necesită precizie mai mare. Poate reprezenta valori aflate în intervalul 1.0×10^{-28} și 7.9×10^{28} , cu 28 de cifre semnificative. Acest tip nu poate reprezenta zero cu semn, infinit sau NaN. Dacă în urma operațiilor, un număr este prea mic pentru a putea fi reprezentat ca decimal, atunci el este făcut 0, iar dacă este prea mare, rezultă o excepție. Diferența principală față de tipurile în virgulă mobilă este că are o precizie mai mare, dar un domeniu de reprezentare mai mic. Din cauza aceasta, nu se fac conversii implicite între nici un tip în virgulă mobilă și decimal și nu este posibilă mixarea variabilelor de acest tip într-o expresie, fără conversii explicite.

Literalii de acest tip se exprimă folosind ca sufix-de-tip-real caracterele m sau M. Dacă valoarea specificată nu poate fi reprezentată prin tipul decimal, apare o eroare la compilare.

Tipul bool

Este folosit pentru reprezentarea valorilor de adevăr true și false. Literalii care se pot folosi sunt true și false. Nu există conversii standard între bool și nici un alt tip.

2.2.3 Tipul enumerare

Tipul enumerare este un tip valoare, construit pentru a permite declararea constantelor înrudite, într-o manieră clară și sigură din punct de vedere al

tipului. Un exemplu este:

```
using System;
public class Draw
{
    public enum LineStyle
    {
        Solid
        Dotted,
        DotDash
    }

    public void DrawLine(int x1, int y1, int x2, int y2,
        LineStyle lineStyle)
    {
        if (lineStyle == LineStyle.Solid)
        {
            //cod desenare linie continua
        }
        else
        if (lineStyle == LineStyle.Dotted)
        {
            //cod desenare linie punctata
        }
        else
        if (lineStyle == LineStyle.DotDash)
        {
            //cod desenare segment linie-punct
        }
        else
        {
            throw new ArgumentException("Invalid line style");
        }
    }
}

class Test
{
    public static void Main()
    {
        Draw draw = new Draw();
    }
}
```

```
        draw.DrawLine(0, 0, 10, 10, Draw.LineStyle.Solid);
        draw.DrawLine(0, 0, 10, 10, (Draw.LineStyle)100);
    }
}
```

Al doilea apel este legal, deoarece valorile care se pot specifica pentru un enum nu sunt limitate la valorile declarate în enum. Ca atare, programatorul trebuie să facă validări suplimentare pentru a determina consistența valorilor. În cazul de față, la apelul de metodă se aruncă o excepție (excepțiile vor fi tratate pe larg într-un curs viitor).

Ca și mod de scriere a enumerărilor, se sugerează folosirea convenției Pascal atât pentru numele tipului cât și pentru numele valorilor conținute, precum și evitarea folosirii prefixelor sau sufixelor pentru acestea.

Enumerările nu pot fi declarate abstracte și nu pot fi derivate. Orice enum este derivat automat din `System.Enum`, care este la rândul lui derivat din `System.ValueType`; astfel, metodele moștenite de la tipurile părinte sunt utilizabile de către orice variabilă de tip enum.

Fiecare tip enumerare care este folosit are un tip de reprezentare⁸, pentru a se cunoaște cât spațiu de memorie trebuie să fie alocat unei variabile de acest tip. Dacă nu se specifică nici un tip de reprezentare (ca mai sus), atunci se presupune implicit tipul `int`. Specificarea unui tip de reprezentare (care poate fi orice tip integral, exceptând tipul `char`) se face prin enunțarea tipului după numele enumerării:

```
enum MyEnum : byte
{
    small,
    large
}
```

Specificarea este folosită atunci când dimensiunea în memorie este importantă, sau când se dorește crearea unui tip de indicator (un tip flag) al cărui număr de stări diferă de numărul de biți alocați tipului `int` (modelare de flag-uri):

```
enum ActionAttributes : ulong
{
    Read = 1,
    Write = 2,
    Delete = 4,
    Query = 8,
```

⁸Engl: underlying type

```

Sync = 16
//etc
}
...
ActionAttributes aa=ActionAttributes.Read|ActionAttributes.Write
| ActionAttributes.Query;
...

```

În mod implicit, valoarea primului membru al unei structuri este 0, și fiecare variabilă care urmează are valoarea mai mare cu o unitate decât precedentă. La dorință, valoarea fiecărei variabile poate fi specificată explicit:

```

enum Values
{
    a = 1,
    b = 2,
    c = a + b
}

```

Următoarele observații se impun relativ la lucrul cu tipul enumerare:

1. valorile specificate ca inițializatori trebuie să fie reprezentabile prin tipul de reprezentare a enumerării, altfel apare o eroare la compilare:

```

enum Out : byte
{
    A = -1
} //eroare semnalata la compilare

```

2. mai mulți membri pot avea aceeași valoare (manevră dictată de semantica tipului construit):

```

enum ExamState
{
    passed = 10,
    failed = 1,
    rejected = failed
}

```

3. dacă pentru un membru nu este dată o valoare, acesta va lua valoarea membrului precedent + 1 (cu excepția primului membru – vezi mai sus)

4. nu se permit referințe circulare:

```
enum CircularEnum
{
    A = B,
    B
} // A depinde explicit de B, B depinde implicit de A
// eroare semnalata la compilare
```

5. este recomandat ca orice tip enum să conțină un membru cu valoarea 0, pentru că în anumite contexte valoarea implicită pentru o variabilă enum este 0, ceea ce poate duce la inconsistențe și bug-uri greu de depanat

```
enum Months
{
    InvalidMonth, // are valoarea implicita 0, fiind primul element
    January,
    February,
    // etc
}
```

Tipurile enumerare pot fi convertite către tipul lor de bază și înapoi, folosind o conversie explicită (cast):

```
enum Values
{
    a = 1,
    b = 5,
    c = 3
}

class Test
{
    public static void Main()
    {
        Values v = (Values)3;
        int ival = (int)v;
    }
}
```

Valoarea 0 poate fi convertită către un enum fără cast:

```

...
MyEnum me;
...
if (me == 0)
{
    //cod
}

```

Următorul cod arată câteva din artificiiile care pot fi aplicate tipului enumerare: obținerea tipului unui element de tip enumerare precum și a tipului clasei de bază, a tipului de reprezentare, a valorilor conținute (ca nume simbolice și ca valori), conversie de la un string la un tip enumerare pe baza numelui, etc. Exemplul este preluat din <FrameworkSDK>\Samples\Technologies\ValueAndEnumTypes.

```

using System;

namespace DemoEnum
{
    class DemoEnum
    {
        enum Color
        {
            Red    = 111,
            Green  = 222,
            Blue   = 333
        }
        private static void DemoEnums()
        {
            Console.WriteLine("\n\nDemo start: Demo of enumerated types.");
            Color c = Color.Red;

            // What type is this enum & what is it derived from
            Console.WriteLine("    The " + c.GetType() + " type is derived from "
                + c.GetType().BaseType);

            // What is the underlying type used for the Enum's value
            Console.WriteLine("    Underlying type: " + Enum.GetUnderlyingType(
                typeof(Color)));

            // Display the set of legal enum values

```

```
Color[] o = (Color[]) Enum.GetValues(c.GetType());
Console.WriteLine("\n    Number of valid enum values: " + o.Length);
for (int x = 0; x < o.Length; x++)
{
    Color cc = ((Color)(o[x]));
    Console.WriteLine("    {0}: Name={1,7}\t\tNumber={2}", x,
        cc.ToString("G"), cc.ToString("D"));
}

// Check if a value is legal for this enum
Console.WriteLine("\n    111 is a valid enum value: " + Enum.IsDefined(
    c.GetType(), 111));    // True
Console.WriteLine("    112 is a valid enum value: " + Enum.IsDefined(
    c.GetType(), 112));    // False

// Check if two enums are equal
Console.WriteLine("\n    Is c equal to Red: " + (Color.Red == c)); //True
Console.WriteLine("    Is c equal to Blue: " + (Color.Blue == c)); //False

// Display the enum's value as a string using different format specifiers
Console.WriteLine("\n    c's value as a string: " + c.ToString("G")); //Red
Console.WriteLine("    c's value as a number: " + c.ToString("D")); //111

// Convert a string to an enum's value
c = (Color) (Enum.Parse(typeof(Color), "Blue"));
try
{
    c = (Color) (Enum.Parse(typeof(Color), "NotAColor")); //Not valid,
    //raises exception
}
catch (ArgumentException)
{
    Console.WriteLine("    'NotAColor' is not a valid value for this enum.");
}

// Display the enum's value as a string
Console.WriteLine("\n    c's value as a string: " + c.ToString("G")); //Blue
Console.WriteLine("    c's value as a number: " + c.ToString("D")); //333

Console.WriteLine("Demo stop: Demo of enumerated types.");
}
```

```
static void Main()
{
    DemoEnums();
}
}
```

2.3 Tablouri

De multe ori se dorește a se lucra cu o colecție de elemente de un anumit tip. O soluție pentru această problemă o reprezintă tablourile. Sintaxa de declarare este asemănătoare cu cea din Java sau C++, dar fiecare tablou este un obiect, derivat din clasa abstractă `System.Array`. Accesul la elemente se face prin intermediul indicilor care încep de la 0 și se termină la numărul de elemente-1; orice depășire a indicilor duce la apariția unei excepții: `System.IndexOutOfRangeException`. O variabilă de tip tablou poate avea valoare de null sau poate să indice către o instanță validă.

2.3.1 Tablouri unidimensionale

Declararea unui tablou unidimensional se face prin plasarea de paranteze drepte între numele tipului tabloului și numele său, ca mai jos⁹:

```
int[] sir;
```

Declararea de mai sus nu duce la alocare de spațiu pentru memorarea șirului; *instanțierea* se poate face ca mai jos:

```
sir = new int[10];
```

Exemplu:

```
using System;
class Unidimensional
{
    public static int Main()
    {
        int[] sir;
        int n;
        Console.Write("Dimensiunea vectorului: ");
```

⁹Spre deosebire de Java, nu se poate modifica locul parantezelor, adică nu se poate scrie: `int sir[]`.

```

    n = Int32.Parse( Console.ReadLine() );
    sir = new int[n];
    for( int i=0; i<sir.Length; i++)
    {
        sir[i] = i * i;
    }
    for( int i=0; i<sir.Length; i++)
    {
        Console.WriteLine(“sir[{0}]={1}”, i, sir[i]);
    }
    return 0;
}
}

```

În acest exemplu se folosește proprietatea¹⁰ `Length`, care returnează numărul *tuturor* elementelor vectorului (lucru vizibil la tablourile multidimensionale rectangulare). De menționat că în acest context *n* și *sir* nu se pot declara la un loc, adică declarații de genul `int[] sir, n;` sau `int n, []sir;` sunt incorecte (prima este corectă din punct de vedere sintactic, dar ar rezulta că *n* este și el un tablou; în al doilea caz, declarația nu este corectă sintactic).

Se pot face inițializări ale valorilor conținute într-un tablou:

```
int[] a = new int[] {1,2,3};
```

sau în forma mai scurtă:

```
int[] a = {1,2,3};
```

2.3.2 Tablouri multidimensionale

C# cunoaște două tipuri de tablouri multidimensionale: rectangulare și neregulate¹¹. Numele lor vine de la forma pe care o pot avea.

Tablouri rectangulare

Tablourile rectangulare au proprietatea că pe fiecare dimensiune se află același număr de elemente. Se declară ca în exemplul de mai jos:

```
int[,] tab;
```

unde `tab` este un tablou rectangular bidimensional. Instanțierea se face:

¹⁰Pentru noțiunea de proprietate, vezi la partea despre clase.

¹¹Engl: jagged arrays.

```
tab = new int[2,3];
```

rezultând un tablou cu 2 linii și 3 coloane. Referirea la elementul aflat pe linia i și coloana j se face cu $tab[i, j]$.

La declararea tabloului se poate face și inițializare:

```
int[,] tab = new int[,] {{1,2},{3,4}};
```

sau, mai pe scurt:

```
int[,] tab = {{1, 2}, {3, 4}};
```

Exemplu:

```
using System;
class Test
{
    public static void Main()
    {
        int[,] tabInm = new int[10,10];
        for( int i=0; i<tabInm.GetLength(0); i++ )
        {
            for( int j=0; j<tabInm.GetLength(1); j++ )
            {
                tabInm[i,j] = i * j;
            }
        }
        for( int i=0; i<tabInm.GetLength(0); i++ )
        {
            for( int j=0; j<tabInm.GetLength(1); j++ )
            {
                Console.WriteLine("{0}*{1}={2}", i, j, tabInm[i,j]);
            }
        }
        Console.WriteLine("tabInm.Length={0}", tabInm.Length);
    }
}
```

După ce se afișează tabla înmulțirii până la 10, se va afișa: `tabInm.Length=100`, deoarece proprietatea `Length` dă numărul total de elemente aflat în tablou (pe toate dimensiunile). Am folosit însă metoda `GetLength(d)` care returnează numărul de elemente aflate pe dimensiunea numărul d (numărarea dimensiunilor începe cu 0).

Determinarea numărului de dimensiuni pentru un tablou rectangular la run-time se face folosind proprietatea Rank a clasei de bază System.Array.

Exemplu:

```
using System;
class Dimensiuni
{
    public static void Main()
    {
        int[] t1 = new int[2];
        int[,] t2 = new int[3,4];
        int[,,,] t3 = new int[5,6,7];
        Console.WriteLine("t1.Rank={0}\nt2.Rank={1}\nt3.Rank={2}",
            t1.Rank, t2.Rank, t3.Rank);
    }
}
```

Pe ecran va apărea:

```
t1.Rank=1
t2.Rank=2
t3.Rank=3
```

Tablouri neregulate

Un tablou neregulat¹² reprezintă un tablou de tabouri. Declararea unui tablou neregulat cu două dimensiuni se face ca mai jos:

```
int[][] tab;
```

Referirea la elementul de indici i și j se face prin `tab[i][j]`.

Exemplu:

```
using System;
class JaggedArray
{
    public static void Main()
    {
        int[][] a = new int[2][];
        a[0] = new int[2];
        a[1] = new int[3];
        for( int i=0; i<a[0].Length; i++)
```

¹²Engl: jagged array

```

    {
        a[0][i] = i;
    }
    for( int i=0; i<a[1].Length; i++)
    {
        a[1][i] = i * i;
    }
    for(int i=0; i<a.Length; i++)
    {
        for( int j=0; j<a[i].Length; j++ )
        {
            Console.Write("{0} ", a[i][j]);
        }
        Console.WriteLine();
    }
    Console.WriteLine("a.Rank={0}", a.Rank);
}
}

```

va scrie pe ecran:

```

0 1
0 1 4
a.Rank=1

```

Ultima linie afișată se explică prin faptul că un tablou neregulat este un vector care conține referințe, deci este unidimensional.

Inițializarea valorilor unui tablou neregulat se poate face la declarare:

```

int[] [] myJaggedArray = new int [] []
{
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};

```

Forma de mai sus se poate prescurta la:

```

int[] [] myJaggedArray = {
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};

```


2.4 Șiruri de caractere

Tipul de date folosit pentru reprezentarea șirurilor de caractere este clasa `System.String` (pentru care se poate folosi aliasul `"string"`; reamintim că este un tip predefinit). Obiectele de acest tip sunt imutabile (caracterele conținute nu se pot schimba, dar pe baza unui șir se poate obține un alt șir). Șirurile pot conține secvențe escape și pot fi de două tipuri: regulate și de tip `"verbatim"`¹³. Șirurile regulate sunt demarcate prin ghilimele și necesită secvențe escape pentru reprezentarea caracterelor escape.

Exemplu:

```
String a = "string literal";
String versuri = "vers1\nvers2";
String caleCompleta = "\\minimax\protect\csharp";
```

Pentru situația în care se utilizează masiv secvențe escape, se pot folosi șirurile `verbatim`. Un literal de acest tip are simbolul `"@"` înaintea ghilimelelor de început. Pentru cazul în care ghilimelele sunt întâlnite în interiorul șirului, ele se vor dubla. Un șir de caractere poate fi reprezentat pe mai multe rânduri fără a folosi caracterul `\n`. Șirurile `verbatim` sunt folosite pentru a face referiri la fișiere sau chei în regiștri, sau pentru prelucrarea fișierelor.

Exemple:

```
String caleCompleta=@"\\minimax\protect\csharp";
//ghilimelele se dubleaza intr-un verbatim string
String s=@"notiunea ""aleator"" se refera...";
//string multilinie reprezentat ca verbatim
String dialog=@"-Alo? Cu ce va ajutam?
-As avea nevoie de o informatie.";
```

Operatorii `"=="` și `"!="` pentru două șiruri de caractere se comportă în felul următor: două șiruri de caractere se consideră egale dacă sunt fie amândouă `null`, fie au aceeași lungime și caracterele de pe aceleași poziții coincid; `"!="` dă negarea relației de egalitate. Pentru a se compara două șiruri din punct de vedere al referințelor, trebuie ca măcar unul dintre obiecte să fie convertit la `object`:

```
class Test
{
    public static void Main()
    {
```

¹³Engl: `verbatim literals`

```

        string a = "Salut", b = a.Clone();
        System.Console.WriteLine("a==b: {0}", a==b);
        System.Console.WriteLine("(object)a==b: {0}",
            (object)a==b);
    }
}

```

La dispozitivul standard de ieșire se va afișa:

```

a==b: true
(object)a==b: false

```

Alt operator care se poate utiliza este "+", reprezentând concatenarea:

```

String s = "Salut" + " " + "lume";
s += "!!!!";

```

Clasa String pune la dispoziție metode pentru: comparare (Compare, CompareOrdinal, CompareTo), căutare (EndsWith, StartsWith, IndexOf, LastIndexOf), modificare (a se înțelege obținerea altor obiecte pe baza celui curent - Concat, CopyTo, Insert, Join, PadLeft, PadRight, Remove, Replace, Split, Substring, ToLower, ToUpper, Trim, TrimEnd, TrimStart)¹⁴. Accesarea unui caracter aflat pe o poziție *i* a unui șir *s* se face prin folosirea parantezelor drepte, cu aceleași restricții asupra indicelui ca și pentru tablouri: *s[i]*.

În clasa object se află metoda ToString() care este suprascrisă în fiecare clasă ale cărei instanțe pot fi tipărite. Pentru obținerea unei reprezentări diferite se folosește metoda String.Format().

Un exemplu de folosire a funcției Split() pentru despărțirea unui șir în funcție de separatori este:

```

using System;
class Class1
{
    static void Main(string[] args)
    {
        String s = "Oh, I hadn't thought of that!";
        char[] x = { ' ', ' ', ' ', ' ' };
        String[] tokens = s.Split( x );
        for(int i=0; i<tokens.Length; i++)
        {
            Console.WriteLine("Token: {0}", tokens[i]);
        }
    }
}

```

¹⁴A se vedea exemplele din MSDN.

```
    }
  }
}
```

va afișa pe ecran:

```
Token: Oh
Token:
Token: I
Token: hadn't
Token: thought
Token: of
Token: that!
```

De remarcat că pentru caracterul apostrof nu este obligatorie secvența escape în cazul șirurilor de caractere. Al doilea lucru care trebuie explicat este că al doilea token este cuvântul vid, care apare între cei doi separatori alăturați: virgula și spațiul. Metoda `Split()` nu face gruparea mai multor separatori, lucru care ar fi de dorit în prezența a doi separatori alăturați. Pentru aceasta se folosesc expresii regulate.

Pentru a lucra cu șiruri de caractere care permit modificarea lor se folosește clasa `StringBuilder`, din namespace-ul `System.Text`.

2.4.1 Expresii regulate

În cazul în care funcțiile din clasa `String` nu sunt suficient de puternice, namespace-ul `System.Text.RegularExpressions` pune la dispoziție o clasă de lucru cu expresii regulate numită `Regex`. Expresiile regulate reprezintă o metodă extrem de facilă de a opera căutări/înlocuiri pe text. Forma expresiilor regulate este cea din limbajul Perl.

Această clasă folosește o tehnică interesantă pentru mărirea performanțelor: dacă programatorul vrea, se scrie o secvență "din mers" pentru a implementa potrivirea expresiei regulate, după care codul este rulat¹⁵.

Exemplul anterior poate fi rescris corect din punct de vedere al funcționalității prin folosirea unei expresii regulate, pentru a prinde și cazul separatorilor multipli adiacenți:

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

¹⁵Codul este scris direct în IL.

```
String s = "Oh, I hadn't thought of that!";  
//separator: virgula, spatiu sau punct si virgula  
//unul sau mai multe, orice combinatie  
Regex regex = new Regex("[, ;]+");  
String[] strs = regex.Split(s);  
for( int i=0; i<strs.Length; i++)  
{  
    Console.WriteLine("Word: {0}", strs[i]);  
}  
}
```

care va produce:

```
Word: Oh  
Word: I  
Word: hadn't  
Word: thought  
Word: of  
Word: that!
```

Curs 3

Clase – generalități. Instrucțiuni. Spații de nume

3.1 Clase – vedere generală

Clasele reprezintă tipuri referință definite de utilizator. O clasă poate să moștenească o singură clasă și poate implementa mai multe interfețe.

Clasele pot conține constante, câmpuri, metode, proprietăți, evenimente, indexatori, operatori, constructori de instanță, destructori, constructori de clasă, tipuri imbricate. Fiecare membru poate conține un nivel de accesare, care controlează gradul de acces la el. O descriere este dată în tabelul 3.1:

Tabelul 3.1: Modificatori de acces ai membrilor unei clase

Accesor	Semnificație
public	Acces nelimitat
protected	Acces limitat la clasa conținătoare sau la tipuri derivate din ea
internal	Acces limitat la acest assembly
protected internal	Acces limitat la acest assembly sau la tipuri derivate din clasă
private	Acces limitat la clasă; modificatorul implicit de acces

```
using System;
```

```
class MyClass
{
    public MyClass()
    {
        Console.WriteLine("Constructor instanta");
    }
    public MyClass( int value )
    {
        MyField = value;
        Console.WriteLine("Constructor instanta");
    }
    public const int MyConst = 12;
    public int MyField = 34;
    public void MyMethod()
    {
        Console.WriteLine("MyClass.MyMethod");
    }
    public int MyProperty
    {
        get
        {
            return MyField;
        }
        set
        {
            MyField = value;
        }
    }
    public int this[int index]
    {
        get
        {
            return 0;
        }
        set
        {
            Console.WriteLine("this[{0}]={1}", index, value);
        }
    }
}
public event EventHandler MyEvent;
public static MyClass operator+(MyClass a, MyClass b)
```

```
{
    return new MyClass(a.MyField + b.MyField);
}
}

class Test
{
    static void Main()
    {
        MyClass a = new MyClass();
        MyClass b = new MyClass(1);
        Console.WriteLine("MyConst={0}", MyClass.MyConst);
        a.MyField++;
        Console.WriteLine("a.MyField={0}", a.MyField);
        a.MyMethod();
        a.MyProperty++;
        Console.WriteLine("a.MyProperty={0}", a.MyProperty);
        a[3] = a[1] = a[2];
        Console.WriteLine("a[3]={0}", a[3]);
        a.MyEvent += new EventHandler(MyHandler);
        MyClass c = a + b;
    }

    static void MyHandler(object Sender, EventArgs e)
    {
        Console.WriteLine("Test.MyHandler");
    }

    internal class MyNestedType
    {}
}
```

Constanta este un membru al unei clase care reprezintă o valoare nemodificabilă, care poate fi evaluată la compilare. Constantele pot depinde de alte constante, atâta timp cât nu se creează dependențe circulare. Ele sunt considerate automat membri statici (dar este interzis să se folosească specificatorul static în fața lor). Ele pot fi accesate exclusiv prin intermediul numelui de clasă (MyClass.MyConst), și nu prin intermediul vreunei instanțe (a.MyConst).

Câmpul este un membru reprezentând o variabilă asociată unui obiect.

Metoda este un membru care implementează un calcul sau o acțiune care poate fi efectuată asupra unui obiect sau asupra unei clase. Metodele statice (care au în antet cuvântul cheie static) sunt accesate prin intermediul numelui de clasă, pe când cele nestatice (metode instanță) sunt apelate prin intermediul unui obiect.

Proprietatea este un membru care dă acces la o caracteristică a unui obiect sau unei clase. Exemplele folosite până acum includeau lungimea unui vector, numărul de caractere ale unui șir de caractere, etc. Sintaxa pentru accesarea câmpurilor și a proprietăților este aceeași. Reprezintă o altă modalitate de implementare a accesoriilor pentru obiecte.

Evenimentul este un membru care permite unei clase sau unui obiect să pună la dispoziția altora notificări asupra evenimentelor. Tipul acestei declarații trebuie să fie un tip delegat. O instanță a unui tip delegat încapsulează una sau mai multe entități apelabile. Exemplu:

```
public delegate void EventHandler(object sender,
                                System.EventArgs e);

public class Button
{
    public event EventHandler Click;
    public void Reset()
    {
        Click = null;
    }
}

using System;
public class Form1
{
    Button Button1 = new Button1();

    public Form1()
    {
        Button1.Click += new EventHandler(Button1_Click);
    }

    void Button1_Click(object sender, EventArgs e )
    {
        Console.WriteLine("Button1 was clicked!");
    }
}
```



```
    }

    public void Disconnect()
    {
        Button1.Click -= new EventHandler(Button1_Click);
    }
}
```

Mai sus clasa `Form1` adaugă `Button1_Click` ca tratare de eveniment¹ pentru evenimentul `Click` al lui `Button1`. În metoda `Disconnect()`, acest event handler este înlăturat.

Operatorul este un membru care definește semnificația (supraîncărcarea) unui operator care se aplică instanțelor unei clase. Se pot supraîncărca operatorii binari, unari și de conversie.

Indexatorul este un membru care permite unui obiect să fie indexat în același mod ca un tablou (pentru programatorii C++: supraîncărcarea operatorului `[]`).

Constructorii instanță sunt membri care implementează acțiuni cerute pentru inițializarea fiecărui obiect.

Destructorul este un membru special care implementează acțiunile cerute pentru a distruge o instanță a unei clase. Destructorul nu are parametri, nu poate avea modificatori de acces, nu poate fi apelat explicit și poate fi apelat automat de către garbage collector.

Constructorul static este un membru care implementează acțiuni necesare pentru a inițializa o clasă, mai exact membrii statici ai clasei. Nu pot avea parametri, nu pot avea modificatori de acces, nu sunt apelat explicit, ci automat de către sistem.

Moștenirea este de tip simplu, iar rădăcina ierarhiei este clasa `object` (alias `System.Object`).

3.2 Transmiterea de parametri

În general, transmiterea parametrilor se face prin valoare. Acest lucru înseamnă că la apelul unei metode în stiva gestionată de compilator se copiază

¹Engl: event handler

valoarea parametrului actual transmis, iar la revenire din metodă această valoare va fi ștearsă. Exemplificăm mai jos acest lucru pentru tipurile valoare și referință.

```
using System;
class DemoTipValoare
{
    static void f(int x)
    {
        Console.WriteLine("la intrare in f: {0}", x );
        ++x;
        Console.WriteLine("la iesire din f: {0}", x );
    }
    static void Main()
    {
        int a = 100;
        Console.WriteLine("inainte de intrare in f: {0}", a);
        f( a );
        Console.WriteLine("dupa executarea lui f: {0}", a);
    }
}
```

Executarea acestui program va avea ca rezultat:

```
inainte de intrare in f: 100
la intrare in f: 100
la iesire din f: 101
dupa executarea lui f: 100
```

Pentru variabile de tip referință, pe stivă se depune tot o copie a valorii obiectului. Însă pentru un asemenea tip de variabilă acest lucru înseamnă că pe stivă se va depune ca valoare adresa de memorie la care este stocat obiectul respectiv. Ca atare, metoda apelată poate să modifice starea obiectului care se transmite, dar nu obiectul în sine (adică referința sa):

```
class Employee
{
    public String name;
    public decimal salary;
}
```

```
class Test
```

```
{
    static void Main()
    {
        Employee e = new Employee();
        e.name = "Ionescu";
        e.salary = 300M;
        System.Console.WriteLine("pre: name={0}, salary={1}",
                                e.name, e.salary );

        Method( e );
        System.Console.WriteLine("post: name={0}, salary={1}",
                                e.name, e.salary );
    }

    static void Method( Employee e )
    {
        e.salary += 100;
    }
}
```

va avea ca rezultat:

```
pre: name=Ionescu, salary=300
post: name=Ionescu, salary=400
```

Totuși, chiar și în cazul tipului referință încercarea de a re-crea în interiorul unei metode un obiect transmis ca parametru nu are nici un efect după terminarea ei:

```
class MyClass
{
    public int x;
}

class Test
{
    static void f(MyClass myClass)
    {
        Console.WriteLine("intrare in f: {0}", myClass.x);
        myClass = new MyClass();
        myClass.x = -100;
        Console.WriteLine("iesire din f: {0}", myClass.x);
    }
}
```

```
static void Main()
{
    MyClass myClass = new MyClass();
    myClass.x = 100;
    Console.WriteLine("inainte de apel: {0}", myClass.x);
    f( myClass );
    Console.WriteLine("dupa apel: {0}", myClass.x);
}
}
```

Ieșirea acestui program va fi:

```
inainte de apel: 100
intrare in f: 100
iesire din f: -100
dupa apel: 100
```

Există situații în care acest comportament nu este cel dorit: am vrea ca efectul asupra unui parametru să se mențină și după ce metoda apelată s-a terminat.

Un parametru *referință* este folosit tocmai pentru a rezolva problema transmiterii prin valoare, folosind referință (un alias) pentru entitatea dată de către metoda apelantă. Pentru a transmite un parametru prin referință, se prefixează cu cuvântul cheie *ref* la apel sau la declarare de metodă:

```
using System;
class Test
{
    static void Swap( ref int a, ref int b)
    {
        int t = a;
        a = b;
        b = t;
    }

    static void Main()
    {
        int x=1, y=2;
        Console.WriteLine("inainte de apel: x={0}, y={1}", x, y);
        Swap( ref a, ref b )
        Console.WriteLine("dupa apel: x={0}, y={1}", x, y);
    }
}
```

va realiza interschimbarea valorilor a și b .

Una din trăsăturile specifice parametrilor referință este că valorile pentru care se face apelul trebuie să fie inițializate. Neasignarea de valori pentru x și y în exemplul de mai sus duce o eroare de compilare. Mai clar, exemplul de mai jos generează eroare la compilare, mesajul fiind: "Use of unassigned local variable 'x'".

```
class TestRef
{
    static void f(ref int x)
    {
        x = 100;
    }
    static void Main()
    {
        int x;
        f(ref x);
    }
}
```

Există cazuri în care dorim să obținem același efect ca la parametrii referință, dar fără a trebui să inițializăm argumentele date de către metoda apelantă (de exemplu când valoarea acestui parametru se calculează în interiorul metodei apelate). Pentru aceasta există parametrii de ieșire², similar cu parametrii referință, cu deosebirea că nu trebuie asignată o valoare parametrului de apel:

```
using System;
class Test
{
    static void Main()
    {
        int l = 10;
        double area;
        ComputeSquareArea( l, out area);
        Console.WriteLine("Area is: {0}", area);
    }

    static void ComputeSquareArea( double l, out double area )
    {
```

²Engl: output parameters

```

        area = 1 * 1;
    }
}

```

Pentru toate tipurile de parametri de mai sus există o mapare 1 la 1 între parametrii actuali și cei formali. Un parametru vector³ permite o relație de tipul unul-la-mulți: mai mulți parametri actuali pot fi referite prin intermediul unui singur parametru formal. Un astfel de parametru se declară folosind modificatorul `params`. Pentru o implementare de metodă, putem avea cel mult un parametru de tip vector și acesta trebuie să fie ultimul în lista de parametri. Acest parametru formal este tratat ca un tablou unidimensional:

```

using System;
class Test
{
    static void F(params int[] args)
    {
        Console.WriteLine("# of parameters: {0}", args.Length);
        for( int i=0; i<args.Length; i++)
        {
            Console.WriteLine("args[{0}]= {1}", i, args[i]);
        }
    }

    static void Main()
    {
        F();
        F(1);
        F(1,2);
        F(new int[] {1,2,3});
    }
}

```

Acest tip de transmitere se folosește și de către metoda `WriteLine` (sau `Write`) a clasei `Console`, i.e. există în această clasă o metodă de tipul:

```

public static void WriteLine(string format, params object[] args){...}

```

³Engl: parameter array

3.3 Conversii

O conversie permite ca o expresie de un anumit tip să fie tratată ca fiind de alt tip. Conversiile pot fi implicite sau explicite, aceasta specificând de fapt dacă un operator de cast (conversie) este sau nu necesar.

3.3.1 Conversii implicite

Sunt clasificate ca și conversii implicite următoarele:

- conversiile identitate
- conversiile numerice implicite
- conversiile implicite de tip enumerare
- conversiile implicite de referințe
- boxing
- conversiile implicite ale expresiilor constante
- conversii implicite definite de utilizator

Conversiile implicite pot apărea într-o varietate de situații, de exemplu apeluri de funcții sau atribuirii. Conversiile implicite predefinite nu determină niciodată apariția de excepții.

Conversiile identitate

O conversie identitate convertește de la un tip oarecare către același tip.

Conversiile numerice implicite

Conversiile numerice implicite sunt:

- de la sbyte la short, int, long, float, double, decimal;
- de la byte la short, ushort, int, uint, long, ulong, float, double, decimal;
- de la short la int, long, double, decimal;
- de la ushort la int, uint, long, ulong, float, double, decimal;
- de la int la long, float, double, decimal;

- de la uint la long, ulong, float, double, decimal;
- de la long la float, double, decimal;
- de la ulong la float, double, decimal;
- de la char la ushort, int, uint, long, ulong, float, double, decimal;
- de la float la double.

Conversiile de la int, uint, long, ulong la float, precum și cele de la long sau ulong la double pot duce la o pierdere a preciziei, dar niciodată la o reducere a ordinului de mărime. Alte conversii numerice implicite niciodată nu duc la pierdere de informație.

Conversiile de tip enumerare implicite

O astfel de conversie permite ca literalul 0 să fie convertit la orice tip enumerare (chiar dacă acesta nu conține valoarea 0) - a se vedea 2.2.3, pag. 29.

Conversii implicite de referințe

Conversiile implicite de referințe implicite sunt:

- de la orice tip referință la object;
- de la orice tip clasă B la orice tip clasă A, dacă B este derivat din A;
- de la orice tip clasă A la orice interfață B, dacă A implementează B;
- de la orice interfață A la orice interfață B, dacă A este derivată din B;
- de la orice tip tablou A cu tipul A_E la un tip tablou B având tipul B_E , cu următoarele condiții:
 1. A și B au același număr de dimensiuni;
 2. atât A_E cât și B_E sunt tipuri referință;
 3. există o conversie implicită de tip referință de la A_E la B_E
- de la un tablou la System.Array;
- de la tip delegat la System.Delegate;
- de la orice tip tablou sau tip delegat la System.ICloneable;
- de la tipul null la orice variabilă de tip referință;

Conversie de tip boxing

Permite unui tip valoare să fie implicit convertit către tipul `object` sau `System.ValueType` sau către orice tip interfață pe care tipul valoare îl implementează. O descriere mai amănunțită este dată în secțiunea 3.3.4.

3.3.2 Conversiile implicite ale expresiilor constante

Permit următoarele tipuri de conversii:

- o expresie constantă de tip `int` poate fi convertită către tipurile `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, cu condiția ca valoarea expresiei constante să se afle în domeniul tipului destinație;
- o expresie constantă de tip `long` poate fi convertită la tipul `ulong`, dacă valoarea ce se convertește nu este negativă.

Conversii implicite definite de utilizator

Constau într-o conversie implicită standard opțională, urmată de execuția unui operator de conversie implicită utilizator urmată de altă conversie implicită standard opțională. Regulile exacte sunt descrise în [5].

3.3.3 Conversii explicite

Următoarele conversii sunt clasificate ca explicite:

- toate conversiile implicite
- conversiile numerice explicite
- conversiile explicite de enumerări
- conversiile explicite de referințe
- unboxing
- conversii explicite definite de utilizator

Din cauză că orice conversie implicită este de asemenea și una explicită, aplicarea operatorului de cast este redundantă:

```
int x = 0;
long y = (long)x;//(long) este redundant
```

Conversii numerice explicite

Sunt conversii de la orice tip numeric la un alt tip numeric pentru care nu există conversie numerică implicită:

- de la sbyte la byte, ushort, uint, ulong, char;
- de la byte la sbyte, char;
- de la short la sbyte, byte, ushort, uint, ulong, char;
- de la ushort la sbyte, byte, short, char;
- de la int la sbyte, byte, short, ushort, int, char;
- de la uint la sbyte, byte, short, ushort, int, uint, long, ulong, char;
- de la long la sbyte, byte, short, ushort, int, uint, ulong, char;
- de la ulong la sbyte, byte, short, ushort, int, uint, long, char;
- de la char la sbyte, byte, short;
- de la float la sbyte, byte, short, ushort, int, uint, long, ulong, decimal;
- de la double la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal;
- de la decimal la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double;

Pentru că în astfel de conversii pot apărea pierderi de informație, există două contexte în care se fac aceste conversii: checked și unchecked.

În context checked, conversia se face cu succes dacă valoarea care se convertește este reprezentabilă de către tipul către care se face conversia. În cazul în care conversia nu se poate face cu succes, se va arunca excepția `System.OverflowException`. În context unchecked, conversia se face întotdeauna, dar se poate ajunge la pierdere de informație sau la valori ce nu sunt bine nedefinite (vezi [5], pag. 115–116).

Conversii explicite de enumerări

Conversiile explicite de enumerări sunt:

- de la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal la orice tip enumerare;

- de la orice tip enumerare la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal;
- de la orice tip enumerare la orice tip enumerare.

Conversiile de tip enumerare se fac prin tratarea fiecărui tip enumerare ca fiind tipul întreg de reprezentare, după care se efectuează o conversie implicită sau explicită între tipuri (ex: dacă se dorește conversia de la un tip enumerare E care are tipul de reprezentare int la un tip byte, se va face o conversie explicită de la int la byte; invers, se va face o conversie implicită de la byte la int).

Conversii explicite de referințe

Conversiile explicite de referințe sunt:

- de la object la orice tip referință;
- de la orice tip clasă A la orice tip clasă B, cu condiția ca A să fie clasă de bază pentru B;
- de la orice tip clasă A la orice tip interfață B, dacă A nu este nederivabilă și A nu implementează pe B;
- de la orice tip interfață A la orice tip clasă B, dacă B nu este nederivabilă sau cu condiția ca B să implementeze A;
- de la orice tip interfață A la orice tip interfață B, dacă A nu este derivat din B;
- de la un tip tablou A cu elemente de tip A_E la un tip tablou B cu elemente B_E , cu condițiile:
 1. A și B au același număr de dimensiuni;
 2. A_E și B_E sunt tipuri referință;
 3. există o conversie de referință explicită de la A_E la B_E
- de la System.Array și interfețele pe care le implementează la orice tip tablou;
- de la System.Delegate și interfețele pe care le implementează la orice tip delegat.

Acest tip de conversii cer verificare la run-time. Dacă o astfel de conversie eșuează, se va arunca o excepție de tipul System.InvalidCastException.

Unboxing

Unboxing-ul permite o conversie explicită de la `object` sau `System.ValueType` la orice tip valoare, sau de la orice tip interfață la orice tip valoare care implementează tipul interfață. Mai multe detalii se vor da în secțiunea 3.3.4.

Conversii explicite definite de utilizator

Constau într-o conversie standard explicită opțională, urmată de execuția unei conversii explicite, urmată de o altă conversie standard explicită opțională.

3.3.4 Boxing și unboxing

Boxing-ul și unboxing-ul reprezintă modalitatea prin care C# permite utilizarea simplă a sistemului unificat de tipuri. Spre deosebire de Java, unde există tipuri primitive (care nu pot conține metode) și tipuri referință, în C# toate tipurile sunt derivate din clasa `object` (alias `System.Object`). De exemplu, tipul `int` (alias `System.Int32`) este derivat din clasa `System.ValueType` care la rândul ei este derivată din clasa `object` (alias `System.Object`). Ca atare, un întreg este compatibil cu `object`.

Boxing

Conversia de tip boxing permite oricărui tip valoare să fie implicit convertit către tipul `object` sau către un tip interfață implementat de tipul valoare. Boxing-ul unei valori constă în alocarea unei variabile de tip `object` și copierea valorii inițiale în acea instanță.

Procesul de boxing al unei valori sau variabile de tip valoare se poate înțelege ca o simulare de creare de clasă pentru acel tip:

```
sealed class T_Box
{
    T value;
    public T_Box(T t)
    {
        value = t;
    }
}
```

Astfel, declarațiile:

```
int i = 123;
object box = i;
```

corespund conceptual la:

```
int i = 123;
object box = new int_Box(i);
```

Pentru secvența:

```
int i = 10; // linia 1
object o = i; // linia 2
int j = (int)o; // linia 3
```

procesul se desfășoară ca în figura 3.1: la linia 1, se declară și se inițializează o variabilă de tip `int` `i`, care va conține valoarea 10. La următoarea linie se va crea o referință `o` către un obiect alocat în heap, care va conține atât valoarea 10, cât și o informație despre tipul de dată conținut (în cazul nostru, `System.Int32`). Unboxing-ul se face printr-un cast explicit, ca în linia a treia.

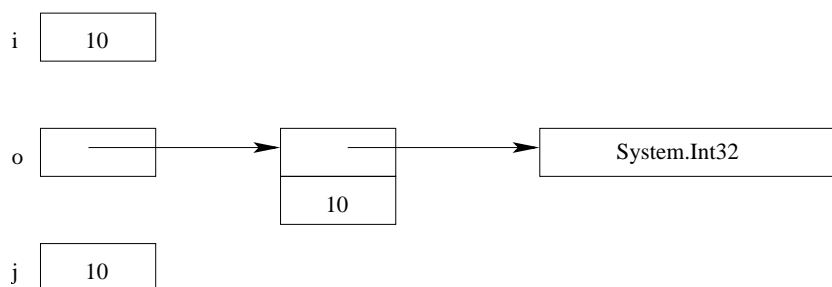


Figura 3.1: Boxing și unboxing

Determinarea tipului pentru care s-a făcut împachetarea se face prin intermediul operatorului `is`:

```
int i = 123;
object o = i;
if (o is int)
{
    Console.WriteLine("Este un int inauntru!");
}
```

Boxing-ul duce la o clonare a valorii care va fi conținută. Altfel spus, secvența:

```
int i = 10;
object o = i;
i++;
Console.WriteLine("in o: {0}", o);
```

va afișa valoarea nealterată 10.

3.3.5 Declarații de variabile și constante

Variabilele și constantele trebuie declarate în C#. Opțional, pentru variabile se poate specifica valoarea inițială, iar pentru constante acest lucru este obligatoriu. O variabilă trebuie să aibă valoarea asignată definită înainte ca valoarea ei să fie utilizată, în cazul în care este declarată în interiorul unei metode. Este o eroare ca într-un sub-bloc să se declare o variabilă cu același nume ca în blocul conținător:

```
void F()
{
    int x = 3, y; //ok
    const double d = 1.1; //ok
    {
        string x = "Mesaj: "; //eroare, x mai este declarat
        //in blocul continator
        int z = x + y; //eroare, y nu are o valoare definita asignata
    }
}
```

Constantele au valori inițiale care trebuie să se poată evalua la compilare.

3.3.6 Declarații de etichete

O etichetă poate prefixa o instrucțiune. Ea este vizibilă în întregul bloc și toate sub-blocurile conținute. O etichetă poate fi referită de către o instrucțiune *goto*:

```
class DemoLabel
{
    int F(int x)
    {
        if (x >= 0) goto myLabel;
        x = -x;
        myLabel: return x;
    }
    static void Main()
    {
        DemoLabel dl = new DemoLabel();
        dl.f();
    }
}
```

3.3.7 Instrucțiuni de selecție

Instrucțiunea if

Instrucțiunea if execută o instrucțiune în funcție de valoarea de adevăr a unei expresii logice. Are formele:

```
if (expresie logica) instructiune;  
if (expresie logica) instructiune; else instructiune;
```

Instrucțiunea switch

Permite executarea unei instrucțiuni în funcție de valoarea unei expresii, care se poate regăsi sau nu într-o listă de valori candidat:

```
switch (expresie)  
{  
    case eticheta: instructiune;  
    case eticheta: instructiune;  
    ...  
    default: instructiune;  
}
```

O etichetă reprezintă o expresie constantă. O instrucțiune poate să și lipsească și în acest caz se va executa instrucțiunea de la *case*-ul următor, sau de la *default*. Secțiunea *default* poate să lipsească. Dacă o instrucțiune este nevidă, atunci obligatoriu va avea la sfârșit o instrucțiune *break* sau *goto case expresieConstanta* sau *goto default*.

Expresia după care se face selecția poate fi de tip *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *string*, *enumerare*. Dacă valoarea expresiei se regăsește printre valorile specificate la clauzele *case*, atunci instrucțiunea corespunzătoare va fi executată; dacă nu, atunci instrucțiunea de la clauza *default* va fi executată (dacă ea există). Spre deosebire de C și C++, e interzis să se folosească fenomenul de "cădere" de la o etichetă la alta; continuarea se face folosind explicit *goto*. Subliniem că se permite folosirea unui selector de tip string.

```
switch (i)  
{  
    case 0:  
        Console.WriteLine("0");  
        break;  
    case 1:
```

```

        Console.Write("Valoarea ");
        goto case 2;
    case 2:
    case 3:
        Console.WriteLine(i);
        break;
    case 4:
        goto default;
    default:
        Console.WriteLine("Numar in afara domeniului admis");
        break;//neaparut, altfel eroare de compilare
}

```

Remarcăm în exemplul de mai sus că chiar și în cazul lui *default* e necesar să se folosească instrucțiune de salt (în cazul nostru *break*); o motivație ar fi că această clauză *default* nu e necesar să fie trecută ultima în *switch*, ci chiar și pe prima poziție – desigur caz mai rar întâlnit.

Există un caz în care *break*, *goto case valoare* sau *goto default* pot să lipsească: când este evident că o asemenea instrucțiune *break/goto* nu ar putea fi atinsă (i.e. sunt prezente instrucțiunile *return*, *throw* sau o ciclare despre care se poate afirma la compilare că este infinită).

3.3.8 Instrucțiuni de ciclare

Există 4 instrucțiuni de ciclare: *while*, *do*, *for*, *foreach*.

Instrucțiunea *while*

Permite executarea unei instrucțiuni atâta timp cât valoarea unei expresii logice este adevărată (ciclu cu test anterior).

Sintaxa:

```
while (expresie logica) instructiune;
```

În interiorul unei astfel de instrucțiuni se poate folosi o instrucțiune de salt de tip *break* sau *continue*.

```

while (r != 0)
{
    r = a%b;
    a = b;
    b = a;
}

```


Instrucțiunea do

Execută o instrucțiune o dată sau de mai multe ori, cât timp o condiție logică este adevărată (ciclu cu test posterior).

Exemplu:

```
do
{
    S += i++;
}while(i<=n)
```

Poate conține instrucțiuni *break* sau *continue*.

Instrucțiunea for

Execută o secvență de inițializare, după care va executa o instrucțiune atâta timp cât o condiție este adevărată (ciclu cu test anterior); poate să conțină un pas de reinițializare (trecerea la pasul următor). Se permite folosirea instrucțiunilor *break* și *continue*.

Exemplu:

```
for (int i=0; i<n; i++)
{
    Console.WriteLine("i={0}", i);
}
```

Instrucțiunea foreach

Enumeră elementele dintr-o colecție, executând o instrucțiune pentru fiecare element. Colecția poate să fie orice instanță a unei clase care implementează interfața *System.Collections.IEnumerable*.

Exemplu:

```
int[] t = {1, 2, 3};
foreach( int x in t)
{
    Console.WriteLine(x);
}
```

Elementul care se extrage este de tip read-only (deci nu poate fi transmis ca parametru *ref* sau *out* și nu se poate aplica un operator care să îi schimbe valoarea).

3.3.9 Instrucțiuni de salt

Permit schimbarea ordinii de execuție a instrucțiunilor. Ele sunt: *break*, *continue*, *goto*, *return*, *throw*.

Instrucțiunea *break*

Produce ieșirea forțată dintr-un ciclu de tip *while*, *do*, *for*, *foreach*.

Instrucțiunea *continue*

Pornește o nouă iterație în interiorul celui mai apropiat ciclu conținător de tip *while*, *do*, *for*, *foreach*.

Instrucțiunea *goto*

Goto permite saltul al o anumită instrucțiune. Are 3 forme:

```
goto eticheta;  
goto case expresieconstanta;  
goto default;
```

Cerința este ca eticheta la care se face saltul să fie definită în cadrul funcției curente și saltul să nu se facă în interiorul unor blocuri de instrucțiuni, deoarece nu se poate reface întotdeauna contextul aceluia bloc.

Se recomandă evitarea utilizării intense a acestui cuvânt cheie, în caz contrar se poate ajunge la fenomenul de "spaghetti code". Pentru o argumentare consistentă a acestei indicații, a se vedea articolul clasic al lui Edsger W. Dijkstra, "Go To Statement Considered Harmful": <http://www.acm.org/classics/oct95/>

Instrucțiunea *return*

Determină cedarea controlului funcției apelante de către funcția apelată. Dacă funcția apelată are tip de retur, atunci instrucțiunea *return* trebuie să fie urmată de o expresie care suportă o conversie implicită către tipul de retur.

3.3.10 Instrucțiunile *try*, *throw*, *catch*, *finally*

Permit tratarea excepțiilor. Vor fi studiate în detaliu la capitolul de excepții.

3.3.11 Instrucțiunile checked și unchecked

Controlează contextul de verificare de depășire a domeniului pentru aritmetica pe întregi și conversii. Au forma:

```
checked
{
    //instructiuni
}
unchecked
{
    //instructiuni
}
```

Verificare se va face la run-time.

3.3.12 Instrucțiunea lock

Obține excluderea mutuală asupra unui obiect pentru executarea unui bloc de instrucțiuni. Are forma:

```
lock (x) instructiuni
```

X trebuie să fie de tip referință (dacă este de tip valoare, nu se face boxing).

3.3.13 Instrucțiunea using

Determină obținerea a unei sau mai multor resurse, execută o instrucțiune și apoi disponibilizează resursa:

```
using ( achizitie de resurse ) instructiune
```

O resursă este o clasă sau o structură care implementează interfața *System.IDisposable*, care include o sigură metodă fără parametri *Dispose()*. Achiziția de resurse se poate face sub formă de variabile locale sau a unor expresii; toate acestea trebuie să fie implicit convertibile la *IDisposable*. Variabilele locale alocate ca resurse sunt read-only. Resursele sunt automat dealocate (prin apelul de *Dispose*) la sfârșitul instrucțiunii (care poate fi bloc de instrucțiuni).

Motivul pentru care există această instrucțiune este unul simplu: uneori se dorește ca pentru anumite obiecte care dețin resurse importante să se apeleze automat metodă *Dispose()* de dealocare a lor, cât mai repede cu putință.

Exemplu:

```
using System;
using System.IO;
class Test
{
    static void Main()
    {
        using( TextWriter w = File.CreateText("log.txt") )
        {
            w.WriteLine("This is line 1");
            w.EriteLine("This is line 2");
        }
    }
}
```

3.4 Spații de nume

În cazul creării de tipuri este posibil să se folosească un același nume pentru tipurile noi create de către dezvoltatorii de soft. Pentru a putea folosi astfel de clase care au numele comun, dar responsabilități diferite, trebuie prevăzută o modalitate de a le adresa în mod unic. Soluția la această problemă este crearea spațiilor de nume⁴ care rezolvă, printr-o adresare completă astfel de ambiguități. Astfel, putem folosi de exemplu clasa Buffer din spațiul System (calificare completă: System.Buffer), alături de clasa Buffer din spațiul de nume Curs3: Curs3.Buffer.

Crearea unui spațiu de nume se face prin folosirea cuvântului namespace:

```
using System;
namespace Curs3
{
    public class Buffer
    {
        public Buffer()
        {
            Console.WriteLine("Bufferul meu!");
        }
    }
}
```

Se pot de asemenea crea spații de nume imbricate. Altfel spus, un spațiu de nume este o colecție de tipuri sau de alte spații de nume.

⁴engl: namespaces

3.4.1 Declarații de spații de nume

O declarație de spațiu de nume constă în cuvântul cheie *namespace*, urmat de identificatorul spațiului de nume și de blocul spațiului de nume, delimitat de acolade. Spațiile de nume sunt implicit publice și acest tip de acces nu se poate modifica. În interiorul unui spațiu de nume se pot utiliza alte spații de nume, pentru a se evita calificarea completă a claselor.

Identificatorul unui spațiu de nume poate fi simplu sau o secvență de identificatori separați prin ”.”. Cea de a doua formă permite definirea de spații de nume imbricate, fără a se imbrica efectiv:

```
namespace N1.N2
{
    class A{}
    class B{}
}
```

este echivalentă cu:

```
namespace N1
{
    namespace N2
    {
        class A{}
        class B{}
    }
}
```

Două declarații de spații de nume cu aceeași denumire contribuie la declararea unui același spațiu de nume:

```
namespace N1.N2
{
    class A{}
}
```

```
namespace N1.N2
{
    class B{}
}
```

este echivalentă cu cele două declarații anterioare.

3.4.2 Directiva using

Directiva using facilitează în primul rând utilizarea spațiilor de nume și a tipurilor definite în acestea; ele nu creează membri noi în cadrul unității de program în care sunt folosite, ci au rol de a ușura referirea tipurilor. Nu se pot utiliza în interiorul claselor, structurilor, enumerărilor.

Exemplu: e mai ușor de înțeles un cod de forma:

```
using System;
class A
{
    static void Main()
    {
        Console.WriteLine("Mesaj");
    }
}
```

decât:

```
class A
{
    static void Main()
    {
        System.Console.WriteLine("Mesaj");
    }
}
```

Directiva using poate fi de fapt folosită atât pentru importuri simbolice, cât și pentru crearea de aliasuri.

Directiva using pentru import simbolic

O directivă using permite importarea simbolică a tuturor tipurilor conținute direct într-un spațiu de nume, i.e. folosirea lor fără a fi necesară o calificare completă. Acest import nu se referă și la spațiile de nume conținute:

```
namespace N1.N2
{
    class A{}
}
namespace N3.N4
{
    class B{};
```

```
}
namespace N5
{
    using N1.N2;
    using N3;
    class C
    {
        A a = null;//ok
        N4.B = null;//Eroare, N4 nu a fost importat
    }
}
```

Importarea de spații de nume nu trebuie să ducă la ambiguități:

```
namespace N1
{
    class A{}
}
namespace N2
{
    class A{}
}
namespace N3
{
    using N1;
    using N2;
    class B
    {
        A a = null;//ambiguitate: N1.A sau N2.A?
    }
}
```

În situația de mai sus, conflictul (care poate foarte ușor în cazul în care se folosesc tipuri produse de dezvoltatori diferiți) poate fi rezolvat de o calificare completă:

```
namespace N3
{
    using N1;
    using N2;
    class B
    {
```

```

    N1.A a1 = null;
    N2.A a2 = null;//ok, nu mai este ambiguitate
}
}

```

Tipurile declarate în interiorul unui spațiu de nume pot avea modificatori de acces *public* sau *internal* (modificatorul implicit). Un tip *internal* nu poate fi folosit prin import în afara assembly-ului, pe când unul *public*, da.

Directiva using ca alias

Introduce un identificator care servește drept alias pentru un spațiu de nume sau pentru un tip.

Exemplu:

```

namespace N1.N2
{
    class A{}
}
namespace N3
{
    using A = N1.N2.A;
    class B
    {
        A a = null;
    }
}

```

Același efect se obține creînd un alias la spațiul de nume N1.N2:

```

namespace N3
{
    using N = N1.N2;
    class B
    {
        N.A a = null;
    }
}

```

Identificatorul dat unui alias trebuie să fie unic, adică în interiorul unui namespace nu trebuie să existe tipuri și aliasuri cu același nume:


```

namespace N3
{
    class A{}
}
namespace N3
{
    using A = N1.N2.A;//eroare, deoarece simbolul A mai este definit
}

```

O directivă alias afectează doar blocul în care este definită:

```

namespace N3
{
    using R = N1.N2;
}
namespace N3
{
    class B
    {
        R.A a = null;//eroare, R nu este definit aici
    }
}

```

adică directiva de alias nu este tranzitivă. Situația de mai sus se poate rezolva prin declararea aliasului în afara spațiului de nume:

```

using R = N1.N2;
namespace N3
{
    class B
    {
        R.A a = null;
    }
}
namespace N3
{
    class C
    {
        R.A b = null;
    }
}

```

Numele create prin directive de alias sunt ascunse de către alte declarații care folosesc același identificator în interiorul unui bloc:

```
using R = N1.N2;
namespace N3
{
    class R{}
    class B
    {
        R.A a;//eroare, clasa R nu are membrul A
    }
}
```

Directivele de alias nu se influențează reciproc:

```
namespace N1.N2{}
namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;
    using R3 = R1.N2;//eroare, R1 necunoscut
}
```

Curs 4

Clase

4.1 Declaraarea unei clase

Declaraarea unei clase se face în felul următor:

`atributeopt modificali-de-clasaopt class identificator clasa-de-bazaopt corp-clasa ;opt`

Modificatorii de clasă sunt:

public - clasele publice sunt accesibile de oriunde; poate fi folosit atât pentru clase interioare, cât și pentru clase care sunt conținute în spații de nume;

internal - se poate folosi atât pentru clase interioare, cât și pentru clase care sunt conținute în spații de nume (este modificatorul implicit pentru clase care sunt conținute în spații de nume). Semnifică acces permis doar în clasa sau spațiul de nume care o cuprinde;

protected - se poate specifica doar pentru clase interioare; tipurile astfel calificate sunt accesibile în clasa curentă sau în cele derivate (chiar dacă clasa derivată face parte din alt spațiu de nume);

private - doar pentru clase interioare; semnifică acces limitat la clasa conținătoare; este modificatorul implicit;

protected internal - folosibil doar pentru clase interioare; tipul definit este accesibil în spațiul de nume curent, în clasa conținătoare sau în tipurile derivate din clasa conținătoare;

new - permis pentru clasele interioare; clasa astfel calificată ascunde un membru cu același nume care este moștenit;

sealed - o clasă sealed nu poate fi moștenită; poate fi clasă interioară sau nu;

abstract - clasa care este incomplet definită și care nu poate fi instanțiată; folosibilă pentru clase interioare sau conținute în spații de nume;

abstract - clasa este definită în mai multe fișiere

4.2 Membrii unei clase

Corpul unei clase se specifică în felul următor:

{ declaratii-de-membri };*opt*

Membrii unei clase sunt împărțiți în următoarele categorii:

- constante
- câmpuri
- metode
- proprietăți
- evenimente
- indexatori
- operatori
- constructori (de instanță)
- destructor
- constructor static
- tipuri

Acestor membri le pot fi atașați modificatorii de acces:

public - membrul este accesibil de oriunde;

protected - membrul este accesabil de către orice membru al clasei conținătoare și de către clasele derivate;

internal - membrul este accesabil doar în assembly-ul curent;

protected internal - reuniunea precedentelor două;

private accesabil doar în clasa conținătoare; este specificatorul implicit.

4.3 Câmpuri

Un câmp reprezintă un membru asociat cu un obiect sau cu o clasă. Modificatorii de câmp care se pot specifica opțional înaintea unui câmp sunt cei de mai sus, la care se adaugă modificatorii *new*, *readonly*, *volatile*, *static*, ce vor fi prezentați mai jos. Pentru orice câmp este necesară precizarea unui tip de date, ce trebuie să aibe gradul de accesibilitate cel puțin cu al câmpului ce se declară. Opțional, câmpurile pot fi inițializate cu valori compatibile. Un astfel de câmp se poate folosi fie prin specificarea numelui său, fie printr-o calificare bazată pe numele clasei sau al unui obiect.

Exemplu:

```
class A
{
    int a;//acces implicit de tip privat
    static void Main()
    {
        A objA = new A();
        objA.a = 1;//se poate accesa in interiorul clasei
    }
}
```

4.3.1 Câmpuri instanțe

Dacă o declarație de câmp nu include modificatorul *static*, atunci acel câmp se va regăsi în orice obiect de tipul clasei curente care va fi instanțiat. Modificări ale acestor câmpuri se vor face independent pentru fiecare obiect. Deoarece un astfel de câmp are o valoare specifică fiecărui obiect, accesarea lui se va face prin calificarea cu numele obiectului:

```
objA.a = 1;
```

(dacă modificatorii de acces permit așa ceva). În interiorul unei instanțe de clasă se poate folosi cuvântul *this*, reprezentând referință la obiectul curent.

4.3.2 Câmpuri statice

Când o declarație de câmp include un specificator *static*, câmpul respectiv nu aparține fiecărei instanțe în particular, ci clasei însăși. Accesarea unui câmp static din exteriorul clasei nu se face exclusiv prin intermediul unui obiect, ci prin numele clasei:

```
class B
{
    public static int V = 3;
    static void Main()
    {
        B.V++; //corect
        V++; //corect
        B b = new B();
        b.V++; //incorect
    }
}
```

Dacă se face calificarea unui astfel de câmp folosind un nume de obiect se semnalează o eroare de compilare.

4.3.3 Câmpuri readonly

Declararea unui câmp de tip *readonly* (static sau nu) se face prin specificarea cuvântului *readonly* în declarația sa:

```
class A
{
    public readonly string salut = "Salut";
    public readonly string nume;
    public class A(string nume)
    {
        this.nume = nume;
    }
}
```

Atribuirea asupra unui câmp de tip *readonly* se poate face doar la declararea sa (ca în exemplu de mai sus) sau prin intermediul unui constructor. Valoarea unor astfel de câmpuri nu e obligatorie a fi cunoscute la compilare.

4.3.4 Câmpuri volatile

Modificatorul "volatile" se poate specifica doar pentru tipurile:

- byte, sbyte, short, ushort, int, uint, char, float, bool;
- un tip enumerare având tipul de reprezentare byte, sbyte, short, ushort, int, uint;

- un tip referință

Pentru câmpuri nevolatile, tehnicile de optimizare care reordonează instrucțiunile pot duce la rezultate neașteptate sau nepredictibile în programe multithreading care accesează câmpurile fără sincronizare (efectuabilă cu instrucțiunea lock). Aceste optimizări pot fi făcute de către compilator, de către sistemul de rulare¹ sau de către hardware. Următoarele tipuri de optimizări sunt afectate în prezența unui modificador volatile:

- citirea unui câmp *volatile* este garantată că se va întâmpla înainte de orice referire la câmp care apare după citire;
- orice scriere a unui câmp *volatile* este garantată că se va petrece după orice instrucțiune anterioară care se referă la câmpul respectiv.

4.3.5 Inițializarea câmpurilor

Pentru fiecare câmp declarat se va asigura o valoare implicită astfel:

- numeric: 0
- bool: false
- char: ‘\0’
- enum: 0
- referință: null

4.4 Constante

O constantă este un câmp a cărui valoare poate fi calculată la compilare. O constantă poate fi prefixată de următorii modificali: *new*, *public*, *protected*, *internal*, *protected internal*, *private*. Cuvantul *new* poate să se combine cu unul din ceilalți 4 modificali de acces. Pentru un câmp constant e obligatoriu să se asigneze o valoare calculabilă la compilare:

```
class A
{
    public const int n=2;
}
```

¹Engl: runtime system

Tipul unei constante poate fi *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *float*, *double*, *decimal*, *bool*, *string*, *enum*, referință. Valoarea care se asignează unei constante trebuie să admită o conversie implicită către tipul constantei. Tipul unei constante trebuie să fie cel puțin la fel de accesibil ca și constanta însăși.

Orice câmp constant este automat un câmp static. Un câmp constant diferă de un câmp static *readonly*: *const*-ul are o valoare cunoscută la compilare, pe când valoarea unui *readonly* poate fi inițializată la runtime în interiorul constructorului (cel mai târziu, de altfel).

4.5 Metode

O metodă este un membru care implementează un calcul sau o acțiune care poate fi efectuată de către un obiect sau o clasă. Antetul unei metode se declară în felul următor:

`attributeopt modificador-de-metodaopt tip-de-retur nume (lista-parametrilor-formaliopt) corp-metoda`

unde modificador-de-metoda poate fi:

- orice modificador de acces
- `new`
- `static`
- `virtual`
- `sealed`
- `override`
- `abstract`
- `extern`

Tipul de retur poate fi orice tip de dată care este cel puțin la fel de accesibil ca și metoda însăși sau `void` (absența informației returnate); nume poate fi un identificator de metodă din clasa curentă sau un identificator calificat cu numele unei interfețe pe care o implementează (*NumeInterfata.numeMetoda*); parametrii pot fi de tip *ref*, *out*, *params*, sau fără nici un calificator; corp-metoda este un bloc cuprins între acolade sau doar caracterul “;” (dacă este vorba de o metodă ce nu se implementează în tipul curent).

Despre calificatorii *virtual*, *override*, *sealed*, *new*, *abstract* se va discuta mai pe larg într-o secțiune viitoare.

4.5.1 Metode statice și nestatice

O metodă se declară a fi statică dacă numele ei este prefixat cu modificatorul de metodă static. O astfel de metodă nu operează asupra unei instanțe anume, ci doar asupra clasei. Este o eroare ca o metodă statică să facă referire la un membru nestatic al unei clase. Apelul unei astfel de metode se face prin *NumeClasa.NumeMetoda* sau direct *NumeMetoda* dacă este apelată din context static al aceleiași clase.

O metodă nestatică nu are cuvântul “static” specificat; ea este apelabilă pentru un obiect anume.

4.5.2 Metode externe

Metodele externe se declară folosind modificatorul extern; acest tip de metode sunt implementate extern, de obicei în alt limbaj decât C#. Deoarece o astfel de metodă nu conține o implementare, corpul acestei metode este “;”.

Exemplu: se utilizează metoda `MessageBox` importată din biblioteca `dll User32.dll`:

```
using System;
using System.Runtime.InteropServices;
class Class1
{
    [DllImport("User32.dll")]
    public static extern int MessageBox(int h, string m, string c,
        int type);
    static void Main(string[] args)
    {
        int retVal = MessageBox(0, "Hello", "Caption", 0);
    }
}
```

4.6 Proprietăți

O proprietate este un membru care permite acces la un atribut al unui obiect sau al unei clase. Exemple de proprietăți sunt: lungimea unui șir, numele unui client, textul conținut într-un control de tip `TextBox`. Proprietățile sunt extensii naturale ale câmpurilor, cu deosebirea că ele nu presupun alocarea de memorie. Ele sunt de fapt niște metode (accesori) care permit citirea sau setarea unor atribute ale unui obiect sau clase; reprezintă modalitatea de scriere a unor metode de tip `get/set` pentru clase sau obiecte.

Declararea unei proprietăți se face astfel:

modificator-de-proprietate_{opt} tip numeproprietate definitie-get_{opt} definitie-set_{opt}
unde:

attribute_{opt} modificator-de-acces_{opt} get corp-get

attribute_{opt} modificator-de-acces_{opt} set corp-set

Modificatorii de acces sunt: *protected*, *internal*, *private*, *protected internal*.

Tipul unei proprietăți specifică tipul de dată ce poate fi accesat, i.e. ce valori vor putea fi atribuite proprietății respective (dacă accesorul de tip *set* a fost definit), respectiv care este tipul valorii returnate de această proprietate (corespunzător accesoriului de tip *get*).

Exemplu:

```
using System;
class Circle
{
    private double radius;
    public double Radius
    {
        get
        {
            return radius;
        }
        set
        {
            radius = value;
        }
    }
    public double Area
    {
        get
        {
            return Math.PI * radius * radius;
        }
        set
        {
            radius = Math.Sqrt(value/Math.PI);
        }
    }
}
```

```
class Test
{
    static void Main()
    {
        Circle c = new Circle();
        c.Radius = 10;
        Console.WriteLine("Area: {0}", c.Area);
        c.Area = 15;
        Console.WriteLine("Radius: {0}", c.Radius)
    }
}
```

Un accesori de tip *get* corespunde unei metode fără parametri, care returnează o valoare de tipul proprietății. Când o proprietate este folosită într-o expresie, accesoriul *get* este apelat pentru a returna valoarea cerută.

Un accesori de tip *set* corespunde unei metode cu un singur parametru de tipul proprietății și tip de retur *void*. Acest parametru implicit al lui *set* este numit întotdeauna *value*. Când o proprietate este folosită ca destinatar într-o atribuire, sau când se folosesc operatorii *++* și *--*, accesoriului *set* i se transmite un parametru care reprezintă noua valoare.

În funcție de prezența sau absența accesoriilor, o proprietate este clasificată după cum urmează:

proprietate read–write, dacă are ambele tipuri de accesorii;

proprietate read–only, dacă are doar accesori de tip *get*; este o eroare de compilare să se facă referire în program la o proprietate în sensul în care s-ar cere operarea cu un accesori de tip *set*;

proprietate write–only, dacă este prezent doar accesoriul de tip *set*; este o eroare de compilare utilizarea unei proprietăți într-un context în care ar fi necesară prezența accesoriului *get*.

Există cazuri în care se dorește ca un accesori să aibă un anumit grad de acces (public, de exemplu), iar celalalt alt tip de acces (e.g. *protected*). Începând cu .NET Framework 2.0, acest lucru este posibil:

```
public class Employee
{
    private string name;
    public Employee(string name)
    {
        this.name = name;
    }
}
```

```

}
public string Name
{
    get { return name; }
    protected set { name = value; }
}
}

```

Într-un asemenea caz, trebuie respectată următoarea regulă: întreaga proprietate trebuie să fie declarată cu grad de acces mai larg decât accesorul pentru care se restricționează gradul de acces.

Demn de menționat este că proprietățile pot fi folosite nu doar pentru a asigura o sintaxă simplă de folosit pentru metodele tradiționale de tip get/set, ci și pentru scrierea controalelor .NET utilizator.

În figura 4.1 este dată reprezentarea unui control utilizator:

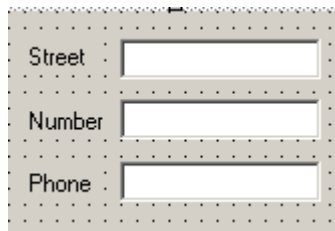


Figura 4.1: Control definit de utilizator

Codul corespunzător este dat mai jos:

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;

namespace UserControlSample
{
    public class UserControl1 : System.Windows.Forms.UserControl
    {
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.TextBox streetTextBox;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.TextBox numberTextBox;
    }
}

```

```
private System.Windows.Forms.Label label3;
private System.Windows.Forms.TextBox phoneTextBox;
private System.ComponentModel.Container components = null;
public UserControl1()
{
    // This call is required by the Windows.Forms Form Designer.
    InitializeComponent();
}

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if( components != null )
            components.Dispose();
    }
    base.Dispose( disposing );
}

#region Component Designer generated code

private void InitializeComponent()
{
    this.label1 = new System.Windows.Forms.Label();
    this.streetTextBox = new System.Windows.Forms.TextBox();
    this.label2 = new System.Windows.Forms.Label();
    this.numberTextBox = new System.Windows.Forms.TextBox();
    this.label3 = new System.Windows.Forms.Label();
    this.phoneTextBox = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    this.label1.AutoSize = true;
    this.label1.Location = new System.Drawing.Point(8, 16);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(34, 13);
    this.label1.TabIndex = 0;
    this.label1.Text = "Street";
    this.streetTextBox.Location = new System.Drawing.Point(56, 14);
    this.streetTextBox.Name = "streetTextBox";
    this.streetTextBox.TabIndex = 1;
    this.streetTextBox.Text = "";
    this.label2.AutoSize = true;
    this.label2.Location = new System.Drawing.Point(8, 48);
```

```

        this.label2.Name = "label2";
        this.label2.Size = new System.Drawing.Size(44, 13);
        this.label2.TabIndex = 2;
        this.label2.Text = "Number";
        this.numberTextBox.Location = new System.Drawing.Point(56, 44);
        this.numberTextBox.Name = "numberTextBox";
        this.numberTextBox.TabIndex = 3;
        this.numberTextBox.Text = "";
        this.label3.AutoSize = true;
        this.label3.Location = new System.Drawing.Point(8, 79);
        this.label3.Name = "label3";
        this.label3.Size = new System.Drawing.Size(37, 13);
        this.label3.TabIndex = 4;
        this.label3.Text = "Phone";
        this.phoneTextBox.Location = new System.Drawing.Point(56, 75);
        this.phoneTextBox.Name = "phoneTextBox";
        this.phoneTextBox.TabIndex = 5;
        this.phoneTextBox.Text = "";
        this.Controls.AddRange(new System.Windows.Forms.Control[] {
            this.phoneTextBox,
            this.label3,
            this.numberTextBox,
            this.label2,
            this.streetTextBox,
            this.label1});
        this.Name = "UserControl1";
        this.Size = new System.Drawing.Size(168, 112);
        this.ResumeLayout(false);
    }
#endregion

[Category ("Data"), Description ("Contents of Street Control")]
public string Street
{
    get{ return streetTextBox.Text; }
    set{ streetTextBox.Text = value; }
}

[Category ("Data"), Description ("Contents of Number Control")]
public string Number
{

```

```

        get{ return numberTextBox.Text; }
        set{ numberTextBox.Text = value; }
    }

    [Category ("Data"), Description ("Contents of Phone Control")]
    public string Phone
    {
        get{ return phoneTextBox.Text; }
        set{ phoneTextBox.Text = value; }
    }
}
}

```

Interesante sunt aici proprietățile publice *Street*, *Number* și *Phone* care vor fi vizibile în fereastra *Properties* atunci când acest control va fi adăugat la o formă. Atributele cuprinse între paranteze drepte sunt opționale, dar vor face ca aceste proprietăți să fie grupate în secțiunea de date a ferestrei *Properties*, și nu în cea “Misc”.

4.7 Indexatori

Uneori are sens tratarea unui obiect ca fiind un vector de elemente. Un indexator este o generalizare a supraîncărcării operatorului `[]` din C++, fiind o facilitare ce dă o mare flexibilitate.

Declararea unui indexator se face în felul următor:

atribute_{opt} modificali-de-indexator_{opt} declarator-de-indexator {declaratii-de-accesori}

Modificatorii de indexator pot fi: *new*, *public*, *protected*, *internal*, *private*, *virtual*, *sealed*, *override*, *abstract*, *extern*. Declaratorul de indexator are forma:

tip-de-retur this[lista-parametrilor-formali].

Lista parametrilor formali trebuie să conțină cel puțin un parametru și nu poate să aibe vreun parametru de tip *ref* sau *out*. Declarațiile de accesori vor conține accesori *get* sau *set*, asemănător cu cei de la proprietăți.

Exemple:

1. Exemplul 1: un indexator simplu:

```

using System;
class MyVector
{

```

```

private double[] v;
public MyVector( int length )
{
    v = new double[ length ];
}
public int Length
{
    get
    {
        return length;
    }
}
public double this[int index]
{
    get
    {
        return v[ index];
    }
    set
    {
        v[index] = value;
    }
}
}

class Test
{
    static void Main()
    {
        MyVector v = new MyVector( 10 );
        v[0] = 0;
        v[1] = 1;
        for( int i=2; i<v.Length; i++)
        {
            v[i] = v[i-1] + v[i-2];
        }
        for( int i=0; i<v.Length; i++)
        {
            Console.WriteLine(“v[“ + i.ToString() + “]=” + v[i]);
        }
    }
}

```



```
}
```

2. Exemplul 2: supraîncărcarea indexatorilor:

```
using System;
using System.Collections;
class DataValue
{
    public DataValue(string name, object data)
    {
        this.name = name;
        this.data = data;
    }
    public string Name
    {
        get
        {
            return(name);
        }
        set
        {
            name = value;
        }
    }

    public object Data
    {
        get
        {
            return(data);
        }
        set
        {
            data = value;
        }
    }
    string name;
    object data;
}

class DataRow
```

```
{
    ArrayList row;
    public DataRow()
    {
        row = new ArrayList();
    }

    public void Load()
    {
        row.Add(new DataValue("Id", 5551212));
        row.Add(new DataValue("Name", "Fred"));
        row.Add(new DataValue("Salary", 2355.23m));
    }

    public object this[int column]
    {
        get
        {
            return(row[column - 1]);
        }
        set
        {
            row[column - 1] = value;
        }
    }

    int FindColumn(string name)
    {
        for (int index = 0; index < row.Count; index++)
        {
            DataValue dataValue = (DataValue) row[index];
            if (dataValue.Name == name)
                return(index);
        }
        return(-1);
    }

    public object this[string name]
    {
        get
        {
```

```

        return this[FindColumn(name)];
    }
    set
    {
        this[FindColumn(name)] = value;
    }
}

class Test
{
    public static void Main()
    {
        DataRow row = new DataRow();
        row.Load();
        DataValue val = (DataValue) row[0];
        Console.WriteLine("Column 0: {0}", val.Data);
        val.Data = 12; // set the ID
        DataValue val = (DataValue) row["Id"];
        Console.WriteLine("Id: {0}", val.Data);
        Console.WriteLine("Salary: {0}",
            ((DataValue) row["Salary"]).Data);
        ((DataValue) row["Name"]).Data = "Barney"; // set the name
        Console.WriteLine("Name: {0}", ((DataValue) row["Name"]).Data);
    }
}

```

3. Exemplul 3: Indexator cu mai mulți parametri:

```

using System;
namespace MyMatrix
{
    class Matrix
    {
        double[,] matrix;

        public Matrix( int rows, int cols )
        {
            matrix = new double[ rows, cols];
        }
    }
}

```

```
public double this[int i, int j]
{
    get
    {
        return matrix[i,j];
    }
    set
    {
        matrix[i,j] = value;
    }
}

public int RowsNo
{
    get
    {
        return matrix.GetLength(0);
    }
}

public int ColsNo
{
    get
    {
        return matrix.GetLength(1);
    }
}

static void Main(string[] args)
{
    MyMatrix m = new MyMatrix(2, 3);
    Console.WriteLine("Lines: {0}", m.RowsNo);
    Console.WriteLine("Columns: {0}", m.ColsNo);
    for(int i=0; i<m.RowsNo; i++)
    {
        for( int j=0; j<m.ColsNo; j++)
        {
            m[i,j] = i + j;
        }
    }
    for(int i=0; i<c.RowsNo; i++)
```

```

        {
            for( int j=0; j<c.ColsNo; j++)
            {
                Console.Write(c[i,j] + " ");
            }
            Console.WriteLine();
        }
    }
}

```

Remarcăm ca accesarea elementelor se face prin perechi de tipul get/set, precum la proprietăți. Ca și în cazul proprietăților, este posibil ca un accesori să aibă un alt grad de acces decât celălalt, folosind același mecanism: se declară indexatorul ca având un anumit grad de accesibilitate, iar pentru un accesori se va declara un grad de acces mai restrictiv.

4.8 Operatori

Un operator este un membru care definește semnificația unei expresii operator care poate fi aplicată unei instanțe a unei clase. Corespunde supraîncărcării operatorilor din C++. O declarație de operator are forma:

`atributeopt modificali-de-operator declaratie-de-operator corp-operator`

Se pot declara operatori unari, binari și de conversie.

Următoarele reguli trebuie să fie respectate pentru orice operator:

1. Orice operator trebuie să fie declarat public și static.
2. Parametrii unui operator trebuie să fie de tip valoare; nu se admite să fie de tip ref sau out.
3. Același modificali nu poate apărea de mai multe ori în antetul unui operator

4.8.1 Operatori unari

Supraîncărcarea operatorilor unari are forma:

`tip operator operator-unar-supraincercabil (tip identificator) corp`

Operatorii unari supraîncărcabili sunt: `+` `-` `!` `~` `++` `-` `true` `false`. Următoarele reguli trebuie să fie respectate la supraîncărcarea unui operator unar (T reprezintă clasa care conține definiția operatorului):

1. Un operator `+`, `-`, `!`, `~` trebuie să preia un singur parametru de tip `T` și poate returna orice tip.
2. Un operator `++` sau `--` trebuie să preia un singur parametru de tip `T` și trebuie să returneze un rezultat de tip `T`.
3. Un operator unar `true` sau `false` trebuie să preia un singur parametru de tip `T` și să returneze `bool`.

Operatorii `true` și `false` trebuie să fie ori ambii definiți, ori nici unul (altfel apare o eroare de compilare). Ei sunt necesari pentru cazuri de genul:

```
if( a==true )
```

sau

```
if( a==false )
```

Mai general, ei pot fi folosiți ca expresii de control în `if`, `do`, `while` și `for`, precum și în operatorul ternar `"? :"`. Deși pare paradoxal, nu este obligatoriu ca `if(a==true)` să fie echivalentă cu `if(!(a==false))`, de exemplu pentru tipuri SQL care pot fi null, ceea ce nu înseamnă nici `true`, nici `false`, ci lipsă de informație.

Exemplu:

```
public struct DBBool
{
    private int x;
    public static bool operator true(DBBool x)
    {
        return x.value > 0;
    }
    public static bool operator false(DBBool x)
    {
        return x.value <= 0;
    }
    ...
}
```

Exemplul de mai jos arată modul în care se face supraîncărcarea operatorului `++`, care poate fi folosit atât ca operator de preincrementare cât și ca operator de postincrementare:

```

public class IntVector
{
    public int Length { ... } // read-only property
    public int this[int index] { ... } // read-write indexer
    public IntVector(int vectorLength) { ... }
    public static IntVector operator++(IntVector iv)
    {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; ++i)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main()
    {
        IntVector iv1 = new IntVector(4); // vector of 4x0
        IntVector iv2;
        iv2 = iv1++; // iv2 contains 4x0, iv1 contains 4x1
        iv2 = ++iv1; // iv2 contains 4x2, iv1 contains 4x2
    }
}

```

4.8.2 Operatori binari

Declararea unui operator binar se face astfel:

tip operator operator-binar-supraincercabil (tip identificator, tip identificator) corp

Operatorii binari supraîncărcabili sunt: + - * / % & | ^ << >> == != > < >= <=.

Cel puțin unul dintre cei doi parametri preluați trebuie să fie de tipul conținător.

Operatorii de shiftare trebuie să aibă primul parametru de tipul clasei în care se declară, iar al doilea parametru de tip int. Unii operatori trebuie să se declare în pereche:

1. operatorii == și !=
2. operatorii > și <
3. operatorii >= și <=

Pentru operatorul `==`, este indicată și definirea metodei `Equals()`, deoarece tipul respectiv va putea fi astfel folosit și de către limbaje care nu suportă supraîncărcarea operatorilor, dar pot apela metoda polimorfică `Equals()` definită în clasa `object`.

Nu se pot supraîncărca operatorii `+`, `=`, `-`, `=`, `/`, `=`, `*`, `=`; dar pentru ca aceștia să funcționeze, este suficient să se supraîncarce operatorii corespunzători: `+`, `-`, `/`, `*`.

4.8.3 Operatori de conversie

O declarație de operator de conversie trebuie introduce o conversie definită de utilizator, care se va adăuga (dar nu va suprascrie) la conversiile predefinite. Declararea unui operator de conversie se face astfel:

implicit operator tip (tip parametru) corp

explicit operator tip (tip parametru) corp

După cum se poate deduce, conversiile pot fi implicite sau explicite. Un astfel de operator va face conversia de la un tip sursă, indicat de tipul parametru-lui din antet la un tip destinație, indicat de tipul de retur. O clasă poate să declare un operator de conversie de la un tip sursă `S` la un tip destinație `T` cu următoarele condiții:

1. `S` și `T` sunt tipuri diferite
2. Unul din cele două tipuri este clasa în care se face definirea.
3. `T` și `S` nu sunt `object` sau tip interfață.
4. `T` și `S` nu sunt baze una pentru cealaltă.

Un bun design asupra operatorilor de conversie are în vedere următoarele:

- Conversiile implicite nu ar trebui să ducă la pierdere de informație sau la apariția de excepții;
- Dacă prima condiție nu este îndeplinită, atunci neapărat trebuie declarată ca o conversie explicită.

Exemplu:

```
using System;
public class Digit
{
    byte value;
    public Digit(byte value)
```



```
{
    if (value < 0 || value > 9) throw new ArgumentException();
    this.value = value;
}
public static implicit operator byte(Digit d)
{
    return d.value;
}
public static explicit operator Digit(byte b)
{
    return new Digit(b);
}
}
```

Prima conversie este implicită pentru că nu va duce la pierderea de informație. Cea de doua poate să arunce o excepție (via constructor) și de aceea este declarată ca și conversie explicită.

4.8.4 Exemplu: clasa Fraction

```
using System;
public class Fraction
{
    public Fraction(int numerator, int denominator)
    {
        Console.WriteLine("In constructor Fraction(int, int)");
        this.numerator=numerator;
        this.denominator=denominator;
    }
    public Fraction(int wholeNumber)
    {
        Console.WriteLine("In Constructor Fraction(int)");
        numerator = wholeNumber;
        denominator = 1;
    }

    public static implicit operator Fraction(int theInt)
    {
        System.Console.WriteLine("In conversie implicita la Fraction");
        return new Fraction(theInt);
    }
}
```

```
public static explicit operator int(Fraction theFraction)
{
    System.Console.WriteLine("In conversie explicita la int");
    return theFraction.numerator /
        theFraction.denominator;
}
public static bool operator==(Fraction lhs, Fraction rhs)
{
    Console.WriteLine("In operator ==");
    if (lhs.denominator * rhs.numerator ==
        rhs.denominator * lhs.numerator )
    {
        return true;
    }
    return false;
}
public static bool operator!=(Fraction lhs, Fraction rhs)
{
    Console.WriteLine("In operator !=");
    return !(lhs==rhs);
}
public override bool Equals(object o)
{
    Console.WriteLine("In metoda Equals");
    if (! (o is Fraction) )
    {
        return false;
    }
    return this == (Fraction) o;
}
public static Fraction operator+(Fraction lhs, Fraction rhs)
{
    Console.WriteLine("In operator+");
    // 1/2 + 3/4 == (1*4) + (3*2) / (2*4) == 10/8
    int firstProduct = lhs.numerator * rhs.denominator;
    int secondProduct = rhs.numerator * lhs.denominator;
    return new Fraction(
        firstProduct + secondProduct,
        lhs.denominator * rhs.denominator
    );
    //ar mai trebui facuta reducerea termenilor
```

```

    }
    public override string ToString( )
    {
        String s = numerator.ToString( ) + "/" +
            denominator.ToString( );
        return s;
    }
    private int numerator;
    private int denominator;
}
public class Tester
{
    static void Main( )
    {
        Fraction f1 = new Fraction(3,4);
        Console.WriteLine("f1: {0}", f1.ToString( ));
        Fraction f2 = new Fraction(2,4);
        Console.WriteLine("f2: {0}", f2.ToString( ));
        Fraction f3 = f1 + f2;
        Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString( ));
        Fraction f4 = f3 + 5;
        Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString( ));
        Fraction f5 = new Fraction(2,4);
        if (f5 == f2)
        {
            Console.WriteLine("F5: {0} == F2: {1}",
                f5.ToString( ),
                f2.ToString( ));
        }
    }
}

```

4.9 Constructori de instanță

Un constructor de instanță este un membru care implementează acțiuni care sunt cerute pentru a inițializa o instanță a unei clase. Declararea unui astfel de constructor se face în felul următor:

atribute_{opt} modificali-de-constructor declarator-de-constructor corp-constructor

Un modificali de constructor poate fi: *public*, *protected*, *internal*, *private*, *extern*. Un declarator de constructor are forma:

nume-clasa (lista-parametrilor-formali_{opt}) initializator-de-constructor_{opt}
unde initializatorul-de-constructor are forma:

```
: base( lista-argumenteopt) sau  
: this( lista-argumenteopt).
```

Corp-constructor poate fi: un bloc de declarații și instrucțiuni delimitat de acolade sau caracterul punct și virgulă.

Un constructor are același nume ca și clasa din care face parte și nu returnează un tip. Constructorii de instanță nu se moștesc. Dacă o clasă nu conține nici o declarație de constructor de instanță, atunci compilatorul va crea automat unul implicit. O clasă care este moștenită dintr-o altă clasă ce nu are constructori fără parametri va trebui să utilizeze un apel de constructor de clasă de bază pentru care să furnizeze parametri potriviți; acest apel se face prin intermediul inițializatorului de constructor. Un constructor poate apela la un alt constructor al clasei din care face parte pentru a efectua inițializări. Când există câmpuri instanță care au o expresie de inițializare în afara constructorilor clasei respective, atunci aceste inițializări se vor face înainte de apelul de constructor al clasei de bază. Pentru a nu se permite crearea sau moștenirea unei clase trebuie ca această clasă să conțină cel puțin un constructor definit de utilizator și toți constructorii să fie declarați privați.

4.10 Constructor static

Un constructor static este un membru care implementează acțiunile cerute pentru inițializarea unei clase. Declararea unui constructor static se face ca mai jos:

```
attributeopt modificador-de-constructor-static identicator( ) corp
```

Modificatorii de constructori statici se pot da sub forma:

```
externopt static sau
```

```
static externopt
```

Constructorii statici nu se moștesc, nu se pot apela direct și nu se pot supra-încărca. Un constructor static se va executa cel mult o dată într-o aplicație. Se garantează faptul că acest constructor se va apela înaintea primei creări a unei instanțe a clasei respective sau înaintea primului acces la un membru static. Acest apel este nedeterminist, necunoscându-se exact când sau dacă se va apela. Un astfel de constructor nu are specificator de acces și poate să acceseze doar membri statici.

Exemplu:

```
class Color  
{  
    public Color(int red, int green, int blue)
```

```
{
    this.red = red;
    this.green = green;
    this.blue = blue;
}
int red;
int green;
int blue;

public static readonly Color Red;
public static readonly Color Green;
public static readonly Color Blue;
// constructor static
static Color()
{
    Red = new Color(255, 0, 0);
    Green = new Color(0, 255, 0);
    Blue = new Color(0, 0, 255);
}
}
class Test
{
    static void Main()
    {
        Color background = Color.Red;
    }
}
```

4.11 Clase interioare

O clasă conține membri, iar în particular aceștia pot fi și clase.

Exemplul 1:

```
using System;
class A
{
    class B
    {
        public static void F()
        {
            Console.WriteLine('A.B.F');
        }
    }
}
```

```

    }
}
static void Main()
{
    A.B.F();
}
}

```

Exemplul 2:

```

public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;
        public Node(object data, Node next)
        {
            this.Data = data;
            this.Next = next;
        }
    }
    private Node first = null;
    private Node last = null;
    //Interfata publica
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}

```

Accesarea unei clase interioare se face prin *NumeClasaExterioara.NumeClasaInterioara* (așa cum este arătat în Exemplul 1), de unde se deduce că o clasă interioară se comportă ca un membru static al tipului conținător. O clasă declarată în interiorul unei alte clase poate avea unul din gradele de accesibilitate *public*, *protected internal*, *protected*, *internal*, *private* (implicit este *private*). O clasă declarată în interiorul unei structuri poate fi declarată *public*, *internal* sau *private* (implicit *private*).

Crearea unei instanțe a unei clase interioare nu trebuie să fie precedată de crearea unei instanțe a clasei exterioare conținătoare, așa cum se vede

din Exemplul 1. O clasă interioară nu are vreo relație specială cu membrul predefinit *this* al clasei conținătoare. Altfel spus, nu se poate folosi *this* în interiorul unei clase interioare pentru a accesa membri instanță din tipul conținător. Dacă o clasă interioară are nevoie să acceseze membri instanță ai clasei conținătoare, va trebui să primească prin constructor parametrul *this* care să se referă la o astfel de instanță:

```
using System;
class C
{
    int i = 123;
    public void F()
    {
        Nested n = new Nested(this);
        n.G();
    }

    public class Nested
    {
        C this_c;
        public Nested(C c)
        {
            this_c = c;
        }
        public void G()
        {
            Console.WriteLine(this_c.i);
        }
    }
}
class Test
{
    static void Main()
    {
        C c = new C();
        c.F();
    }
}
```

Se observă de mai sus că o clasă interioară poate manipula toți membrii care sunt accesibili în interiorul clasei conținătoare, indiferent de gradul lor de

accesibilitate. În cazul în care clasa exterioară (conținătoare) are membri statici, aceștia pot fi utilizați fără a se folosi numele clasei conținătoare:

```
using System;
class C
{
    private static void F()
    {
        Console.WriteLine("C.F");
    }
    public class Nested
    {
        public static void G()
        {
            F();
        }
    }
}
class Test
{
    static void Main()
    {
        C.Nested.G();
    }
}
```

Clasele interioare se folosesc intens în cadrul containerilor pentru care trebuie să se construiască un enumerator. Clasa interioară va fi în acest caz strâns legată de container și va duce la o implementare ușor de urmărit și de întreținut.

4.12 Destructori

Managementul memoriei este făcut sub platforma .NET în mod automat, de către garbage collector, parte componentă a CLR-ului.

În general, acest garbage collector scutește programatorul de grija dealocării memoriei. Dar în anumite situații, se dorește să se facă management manual al dealocării resurselor (de exemplu al resurselor care țin de sistemul de operare: fișiere, conexiuni la rețea sau la baza de date, ferestre, etc, sau al altor resurse al căror management nu se face de către CLR). În C# există

posibilitatea de a lucra cu destructori, sau cu metode de tipul *Dispose()*, *Close()*.

Un destructor se declară în felul următor:

```
attributeopt externopt ~identificator() corp-destructor
```

unde identificator este numele clasei. Un destructor nu are modificador de acces, nu poate fi apelat manual, nu poate fi supraîncărcat, nu este moștenit.

Un destructor este o scurtătură sintactică pentru metoda *Finalize()*, care este definită în clasa *System.Object*. Programatorul nu poate să suprascrie sau să apeleze această metodă.

Exemplu:

```
~MyClass()  
{  
    // Perform some cleanup operations here.  
}
```

Metoda de mai sus este automat translatată în:

```
protected override void Finalize()  
{  
    try  
    {  
        // Perform some cleanup operations here.  
    }  
    finally  
    {  
        base.Finalize();  
    }  
}
```

Problema cu destructorul este că el e chemat doar de către garbage collector, dar acest lucru se face nedeterminist (cu toate că apelarea de destructor se face în cele din urmă, dacă programatorul nu împiedică explicit acest lucru).

Există cazuri în care programatorul dorește să facă dealocarea manual, astfel încât să nu aștepte ca garbage collectorul să apeleze destructorul. Programatorul poate scrie o metoda care să facă acest lucru. Se sugerează definirea unei metode *Dispose()* care ar trebui să fie explicit apelată atunci când resurse de sistem de operare trebuie să fie eliberate. În plus, clasa respectivă ar trebui să implementeze interfața *System.IDisposable*, care conține această metodă.

În acest caz, *Dispose()* ar trebui să inhibe executarea ulterioară de garbage collector pentru instanța curentă. Această manevră permite evitarea eliberării

unei resurse de două ori. Dacă clientul nu apelează explicit *Dispose()*, atunci garbage collectorul va apela el destructorul la un moment dat. Întrucât utilizatorul poate să nu apeleze *Dispose()*, este necesar ca tipurile care implementează această metodă să definească de asemenea destructor.

Exemplu:

```
public class ResourceUser: IDisposable
{
    public void Dispose()
    {
        hwnd.Release();//elibereaza o fereastră in Win32
        GC.SuppressFinalization(this);//elimina apel de Finalize()
    }

    ~ResourceUser()
    {
        hwnd.Release();
    }
}
```

Pentru anumite clase C# se pune la dispoziție o metodă numită *Close()* în locul uneia *Dispose()*: fișiere, socket-uri, ferestre de dialog, etc. Este indicat ca să se adauge o metodă *Close()* care să facă doar apel de *Dispose()*:

```
//in interiorul unei clase
public void Close()
{
    Dispose();
}
```

Modalitatea cea mai indicată este folosirea unui bloc *using*, caz în care se va elibera obiectul alocat (via metoda *Dispose()*) la sfârșitul blocului:

```
using( obiect )
{
    //cod
} //aici se va apela automat metoda Dispose()
```

Curs 5

Clase (2). Moștenire. Interfețe. Structuri

5.1 Clase statice

Începând cu versiunea 2.0 a platformei .NET s-a introdus posibilitatea de a defini clase statice. Acest tip de clasă se folosește atunci când se dorește accesarea membrilor fără a fi nevoie să se lucreze cu obiecte; se pot folosi acolo unde buna funcționare nu este dependentă de starea unor instanțe.

Pentru a crea o clasă statică se folosește cuvântul `static` în declarația de clasă:

```
static class MyStaticClass
{
    //membri statici
}
```

Orice clasă statică are următoarele proprietăți:

1. nu poate fi instanțiată
2. nu poate fi moștenită (este automat *sealed*, vezi secțiunea 5.3)
3. conține doar membri statici

Exemplu:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
```

```

        double celsius = Double.Parse(temperatureCelsius);
        double fahrenheit = (celsius * 9 / 5) + 32;
        return fahrenheit;
    }
    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        double fahrenheit = Double.Parse(temperatureFahrenheit);
        Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;
        return celsius;
    }
}
class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Optiuni");
        Console.WriteLine("1. Celsius->Fahrenheit.");
        Console.WriteLine("2. Fahrenheit->Celsius.");
        Console.WriteLine(":");
        string selection = Console.ReadLine();
        double f, c = 0;
        switch (selection)
        {
            case "1":
                Console.WriteLine("Temperatura Celsius: ");
                f = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperatura in Fahrenheit: {0:F2}", f);
                break;
            case "2":
                Console.WriteLine("Temperatura Fahrenheit: ");
                c = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", c);
                break;
        }
    }
}

```

5.2 Specializarea și generalizarea

Specializarea reprezintă o tehnică de a obține noi clase pornind de la cele existente. Deseori între clasele pe care le modelăm putem observa relații de genul “este un/o”: un om este un mamifer, un salariat este un angajat, etc. Toate acestea duc la crearea unei ierarhii de clase, în care din clase de bază (mamifer sau angajat) descind alte clase, care pe lângă comportament din clasa de bază mai au și caracteristici proprii. Obținerea unei clase derivate plecând de la altă clasă se numește specializare iar operația inversă se numește generalizare. O clasă de bază definește un tip comun, compatibil cu oricare din clasele derivate (direct sau indirect).

În C# o clasă nu trebuie să moștenească explicit din altă clasă; în acest caz se va considera că ea este implicit derivată din clasa predefinită *object* (tot una cu *Object*). C# nu permite moștenire multiplă, eliminând astfel complicațiile întâlnite în C++. Ca alternativă, se permite totuși implementarea de mai multe interfețe (la fel ca în Java).

5.2.1 Specificarea moștenirii

În C# se pentru o clasă *D* se definește clasa de bază *B* folosind următoarea formulă:

```
class D: B
{
    //declaratii si instructiuni
}
```

Dacă pentru o anumită clasă nu se specifică două puncte urmate de numele unei clase de bază atunci *object* va deveni bază pentru clasa în cauză.

Exemplu:

```
//clasa de baza in C#
public class Employee
{
    protected string name;
    protected string ssn;
}

//clasa derivata in C#
public class Salaried : Employee
{
    protected double salary;
```

```

public Salaried( string name, string ssn, double salary )
{
    this.name = name;
    this.ssn = ssn;
    this.salary = salary;
}
}

```

Se observă că câmpurile *name* și *ssn* din clasa de bază sunt accesibile în clasa derivată, datorită specificatorului de acces *protected*.

5.2.2 Apelul constructorilor din clasa de bază

În exemplul anterior nu s-a definit nici un constructor în clasa de bază *Employee*; constructorul clasei derivate trebuie să facă inițializările câmpurilor în conformitate cu parametrii transmiși, chiar dacă o parte din aceste câmpuri provin din clasa de bază. Mai logic ar fi ca în clasa de bază să se găsească un constructor care să inițializeze câmpurile proprii: *name* și *ssn*. Întrucât constructorii nu se moștenesc, e nevoie ca în clasa derivată să se facă un apel explicit al constructorului clasei de bază. Acest apel se face prin *inițializator de constructor* care are forma: două puncte urmate de *base(parametrii-efectivi)*.

```

public class Employee
{
    protected string name;
    protected string ssn;
    public Employee( string name, string ssn)
    {
        this.name = name;
        this.ssn = ssn;
        System.Console.WriteLine("Employee constructor: {0}, {1}",
            name, ssn);
    }
}
public class Salaried : Employee
{
    protected double salary;
    public Salaried(string name, string ssn, double salary):
        base(name, ssn)
    {
        this.salary = salary;
        System.Console.WriteLine("Salaried constructor: {0}",

```

```
        salary);
    }
}
class Test
{
    Salaried s = new Salaried("Jesse", "1234567890",
        100000.00);
}
```

La rulare se va obține:

Employee constructor: Jesse, 1234567890

Salaried constructor: 100000.00

de unde se deduce că apelul de constructor de clasă de bază se face înaintea executării oricăror alte instrucțiuni conținute în constructorul clasei derivate.

Dacă o clasă de bază nu are definit nici un constructor, atunci se va crea unul implicit (fără parametri). Dacă după un constructor al unei clase derivate nu se specifică un inițializator de constructor, atunci va fi apelat constructorul implicit (fie creat automat de compilator, fie scris de către programator); dacă nu există nici un constructor implicit în clasa de bază, atunci programatorul trebuie să specifice un constructor din clasa de bază care va fi apelat, împreună cu parametrii adecvați.

5.2.3 Operatorii *is* și *as*

Operatorul *is*

Operatorul *is* este folosit pentru a verifica dacă un anumit obiect este de un anumit tip. Este folosit de obicei înainte operațiilor de downcasting (conversie explicită de la un tip de bază la unul derivat). Operatorul se folosește astfel:

```
instanta is NumeClasa
```

rezultatul acestei operații fiind true sau false.

Exemplu:

```
Employee e = ...;
if (e is Salaried)
{
    Salaried s = (Salaried)e;
}
```

În cazul în care s-ar face conversia explicită iar obiectul nu este de tipul la care se face conversia ar rezulta o excepție: `System.InvalidCastException`.

Operatorul *as*

Acest operator este folosit pentru conversii explicite, returnând un obiect de tipul la care se face conversia sau null dacă conversia nu se poate face (nu se aruncă excepții). Determinarea validității conversiei se face testând valoarea rezultată față de null: dacă rezultatul e null atunci conversia nu s-a putut face. Ca și precedentul operator se folosește în special la downcasting.

Exemplu:

```
Employee e = ...;
Salaried s = e as Salaried;
if (s != null)
{
    //se lucreaza cu instanta valida de tip Salaried
}
```

5.3 Clase *sealed*

Specificatorul *sealed* care se poate folosi înaintea cuvântului cheie *class* specifică faptul că clasa curentă nu se poate deriva. Este o eroare de compilare ca o clasă *sealed* să fie declarată drept clasă de bază.

5.4 Polimorfismul

Polimorfismul este capacitatea unei entități de a lua mai multe forme. În limbajul C# polimorfismul este de 3 feluri: parametric, ad-hoc și de moștenire.

5.4.1 Polimorfismul parametric

Este cea mai slabă formă de polimorfism, fiind regăsită în majoritatea altor limbaje, ce nu sunt orientate pe obiecte: Pascal, C. Prin polimorfismul parametric se permite ca o implementare de funcție să poată prelucra orice număr de parametri. Acest lucru se poate obține prin folosirea unui parametru de tip *params* (vezi 3.2).

5.4.2 Polimorfismul ad-hoc

Se mai numește și supraîncărcarea metodelor, mecanism prin care în cadrul unei clase se pot scrie mai multe metode, având același nume, dar

tipuri și numere diferite de parametri de apel. Alegerea funcției care va fi apelată se va face la compilare, pe baza corespondenței între tipurile de apel și cele formale.

5.4.3 Polimorfismul de moștenire

Este forma cea mai evoluată de polimorfism. Dacă precedentele forme de polimorfism sunt aplicabile fără a se pune problema de moștenire, în acest caz este necesar să existe o ierarhie de clase. Mecanismul se bazează pe faptul că o clasă de bază definește un tip care este compatibil din punct de vedere al atribuirii cu orice tip derivat, ca mai jos:

```
class B{...}
class D: B{...}
class Test
{
    static void Main()
    {
        B b = new D();//upcasting=conversie implicita catre baza B
    }
}
```

Într-un astfel de caz se pune problema: ce se întâmplă cu metodele având aceeași listă de parametri formali și care se regăsesc în cele două clase?

Să considerăm exemplul următor: avem o clasă *Shape* care conține o metodă *public void Draw()*; din *Shape* se derivează clasa *Polygon* care implementează aceeași metodă în mod specific. Problema care se pune este cum se rezolvă un apel al metodei *Draw* în context de upcasting:

```
class Shape
{
    public void Draw()
    {
        System.Console.WriteLine('Shape.Draw()');
    }
}
class Polygon: Shape
{
    public void Draw()
    {
        System.Console.WriteLine('Polygon.Draw()');
        //desenarea s-ar face prin GDI+
    }
}
```

```

    }
}
class Test
{
    static void Main()
    {
        Polygon p = new Polygon();
        Shape s = p;//upcasting
        s.Draw();
        p.Draw();
    }
}

```

La compilarea acestui cod se va obține un avertisment:

```
warning CS0108: The keyword new is required on Polygon.Draw()
because it hides inherited member Shape.Draw()
```

dar despre specificatorul *new* vom vorbi mai jos (oricum, adăugarea lui nu va schimba cu nimic comportamentul de mai jos, doar va duce la dispariția de avertisment). Codul de mai sus va afișa:

```

Shape.Draw()
Polygon.Draw()

```

Dacă cea de-a doua linie afișată este conformă cu intuiția, primul rezultat este discutabil, dar justificabil: apelul de metodă *Draw()* este rezolvat în fiecare caz la compilare pe baza tipului declarat al obiectelor; ca atare apelul precedent este legat de corpul metodei *Draw* din clasa *Shape*, chiar dacă *s* a fost instanțiat de fapt pe baza unui obiect de tip *Polygon*.

Este posibil ca să se dorească schimbarea acestui comportament: apelul de metodă *Draw* să fie rezolvat în funcție de tipul efectiv al obiectului care face acest apel, și nu de tipul formal declarat. În cazul precedent, apelul *s.Draw()* trebuie să se rezolve de fapt ca fiind către metoda *Draw()* din *Polygon*, pentru că acesta este tipul la rulare al obiectului *s*. Cu alte cuvinte, apelul ar trebui să fie rezolvat la rulare și nu la compilare, în funcție de natura obiectelor. Acest comportament polimorfic este referit sub denumirea *polimorfism de moștenire*.

5.4.4 *Virtual și override*

Pentru a asigura faptul că legarea apelului de metode se face la rulare și nu la compilare, e necesar ca în clasa de bază să se specifice că metoda

Draw() este virtuală, iar în clasa derivată pentru aceeași metodă trebuie să se spună că este o suprascriere a celei din bază:

```
class Shape{
    public virtual void Draw(){...}
}
class Polygon{
    public override void Draw(){...}
}
```

În urma executării metodei *Main* din clasa de mai sus, se va afișa:

```
Polygon.Draw()
Polygon.Draw()
```

adică s-a apelat metoda corespunzătoare tipului efectiv de la rulare, în fiecare caz.

În cazul în care clasa *Polygon* este la rândul ei moștenită și se dorește ca polimorfismul să funcționeze în continuare va trebui ca în această a treia clasă să suprascrie (*override*) metoda *Draw()*.

Un astfel de comportament polimorfic este benefic atunci când se folosește o colecție de obiecte de tipul unei clase de bază:

```
Shape[] painting = new Shape[10];
painting[0] = new Shape();
painting[1] = new Polygon();
...
foreach( Shape s in painting)
    s.Draw();
```

5.4.5 Modificatorul *new* pentru metode

Modificatorul *new* se folosește pentru a indica faptul că o metodă dintr-o clasă derivată care are aceeași semnătură cu una dintr-o clasă de bază nu este o suprascriere a ei, ci o nouă metodă. Este ca și cum metoda declarată *new* ar avea o semnătură diferită.

Să presupunem următorul scenariu: compania A crează o clasă A care are forma:

```
public class A{
    public void M(){
        Console.WriteLine('A.M()');
    }
}
```

O altă companie B va crea o clasă *B* care moștenește clasa A. Compania B nu are nici o influență asupra companiei A (sau asupra modului în care aceasta va face modificări asupra clasei A). Ea va defini în interiorul clasei B o metodă *M()* și una *N()*:

```
class B: A{
    public void M(){
        Console.WriteLine('B.M()');
        N();
        base.M();
    }
    protected virtual void N(){
        Console.WriteLine('B.N()');
    }
}
```

Atunci când compania B compilează codul, compilatorul C# va produce următorul avertisment:

```
warning CS0108: The keyword new is required on 'B.M()' because
it hides inherited member 'A.M()'
```

Acest avertisment va notifica programatorul că clasa *B* definește o metodă *M()*, care va ascunde metoda *M()* din clasa de bază A. Această nouă metodă ar putea schimba înțelesul (semantica) lui *M()*, așa cum a fost creat inițial de compania A. Este de dorit în astfel de cazuri compilatorul să avertizeze despre posibile nepotriviri semantice. Programatorii din B vor trebui să pună în orice caz specificatorul *new* înaintea metodei *B.M()*.

Să presupunem că o aplicație folosește clasa *B()* în felul următor:

```
class App{
    static void Main(){
        B b = new B();
        b.M();
    }
}
```

La rulare se va afișa:

```
B.M()
B.N()
A.M()
```

Să presupunem că A decide adăugarea unei metode virtuale $N()$ în clasa sa, metodă ce va fi apelată din $M()$:

```
public class A
{
    public void M()
    {
        Console.WriteLine('A.M()');
        N();
    }
    protected virtual void N()
    {
        Console.WriteLine('A.N()');
    }
}
```

La o recompilare făcută de B, este dat următorul avertisment:

```
warning CS0114: 'B.N()' hides inherited member 'A.N()'. To make
the current member override that implementation, add the
override keyword. Otherwise, add the new keyword.
```

În acest mod compilatorul avertizează că ambele clase oferă o metodă $N()$ a căror semantică poate să difere. Dacă B decide că metodele $N()$ nu sunt semantic legate în cele două clase, atunci va specifica *new*, informând compilatorul de faptul că versiunea sa este una nouă, care nu suprascrie polimorfic metoda din clasa de bază.

Atunci când codul din clasa *App* este rulat, se va afișa la ieșire:

```
B.M()
B.N()
A.M()
A.N()
```

Ultima linie afișată se explică tocmai prin faptul că metoda $N()$ din *B* este declarată *new* și nu *override* (dacă ar fi fost *override*, ultima linie ar fi fost *B.N()*, din cauza polimorfismului).

Se poate ca B să decidă că metodele $M()$ și $N()$ din cele două clase sunt legate semantic. În acest caz, ea poate șterge definiția metodei *B.M*, iar pentru a semnală faptul că metoda *B.N()* suprascrie metoda omonimă din clasa părinte, va înlocui cuvântul *new* cu *override*. În acest caz, metoda *App.Main* va produce:

A.M()

B.N()

ultima linie fiind explicată de faptul că *B.N()* suprascrie o metodă virtuală.

5.4.6 Metode *sealed*

O metodă de tip *override* poate fi declarată ca fiind de tip *sealed*, astfel împiedicându-se suprascrierea ei într-o clasă derivată din cea curentă:

```
using System;
class A
{
    public virtual void F()
    {
        Console.WriteLine("A.F()");
    }
    public virtual void G()
    {
        Console.WriteLine("A.G()");
    }
}
class B: A
{
    sealed override public void F()
    {
        Console.WriteLine("B.F()");
    }
    override public void G()
    {
        Console.WriteLine("B.G()");
    }
}
class C: B
{
    override public void G()
    {
        Console.WriteLine("C.G()");
    }
}
```

Modificatorul *sealed* pentru *B.F* va împiedica tipul *C* să suprascrie metoda *F*.

5.4.7 Exemplu folosind *virtual*, *new*, *override*, *sealed*

Să presupunem următoare ierarhie de clase, reprezentată în Fig. 5.1; o clasă X moșteneste o clasă Y dacă sensul săgeții este de la X la Y. Fiecare clasă are o metodă *void f()* care determină afișarea clasei în care este definită și pentru care se vor specifica *new*, *virtual*, *override*, *sealed*. Să presupunem

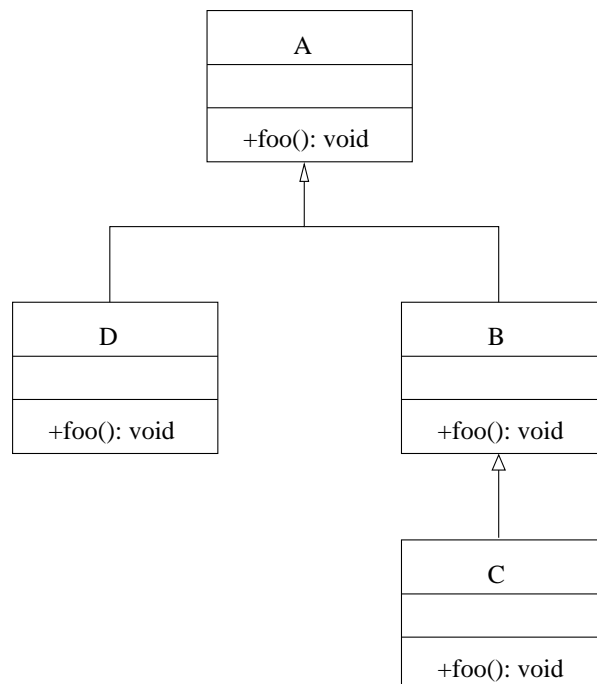


Figura 5.1: Ierarhie de clase

că clasa de test arată astfel:

```

public class Test
{
    static void Main()
    {
        A[] x = new A[4];
        x[0] = new A();
        x[1] = new B();
        x[2] = new C();
        x[3] = new D();

        A a = new A();
        B b = new B();
    }
}
  
```

```

C c = new C();
D d = new D();

/* secventa 1 */
for(int i=0; i<4; i++)
{
    x[i].f();
}
/* secventa 2 */
a.f();
b.f();
c.f();
d.f();
}
}

```

În funcție de specificatorii metodelor $f()$ din fiecare clasă, se obțin ieșirile din tabelul 5.1:

Tabelul 5.1: Efecte ale diferiților specificatori.

Metoda	A.f()	B.f()	C.f()	D.f()
Specificator	virtual	override	override	override
Ieșire secv. 1	A.f	B.f	C.f	D.f
Ieșire secv. 2	A.f	B.f	C.f	D.f
Specificator	virtual	override	new	override
Ieșire secv. 1	A.f	B.f	B.f	D.f
Ieșire secv. 2	A.f	B.f	C.f	D.f
Specificator	virtual	new	new	new
Ieșire secv. 1	A.f	A.f	A.f	A.f
Ieșire secv. 2	A.f	B.f	C.f	D.f
Specificator	virtual	new	override	override
Eroare de compilare deoarece C.f nu poate suprascrie metoda nevirtuală B.f()				
Specificator	virtual	virtual	override	override
Ieșire secv. 1	A.f	A.f	A.f	D.f
Ieșire secv. 2	A.f	B.f	C.f	D.f
Avertisment la compilare deoarece				

Tabelul 5.1 (continuare)

Metoda	A.f()	B.f()	C.f()	D.f()
B.f înlocuiește A.f				
Specificator	virtual	sealed override	override	override
Eroare de compilare deoarece deoarece B.f nu poate fi suprascrisă de C.f				

5.5 Clase și metode abstracte

Deseori pentru o anumită clasă nu are sens crearea de instanțe, din cauza unei generalități prea mari a tipului respectiv. Spunem că această clasă este *abstractă*, iar pentru a împiedica efectiv crearea de instanțe de acest tip, se va specifica cuvântul *abstract* înaintea metodei. În exemplele de mai sus, clasele *Employee* și *Shape* ar putea fi gândite ca fiind abstracte: ele conțin prea puțină informație pentru a putea crea instanțe utile.

Analog, pentru o anumită metodă din interiorul unei clase uneori nu se poate specifica o implementare. De exemplu, pentru clasa *Shape* de mai sus, este imposibil să se dea o implementare la metoda *Draw()*, tocmai din cauza generalității acestei clase. Ar fi util dacă pentru această metodă programatorul ar fi obligat să dea implementări specifice ale acestei metode pentru diversele clase derivate. Pentru a se asigura tratarea polimorfică a acestui tip abstract, orice metodă abstractă este automat și virtuală. Orice metodă care este declarată abstractă implică declararea clasei ca fiind abstractă.

Exemplu:

```
abstract class Shape
{
    public abstract void Draw();
    //remarcam lipsa implementarii si semnul punct si virgula
}
```

Orice clasă care este derivată dintr-o clasă abstractă va trebui fie să nu aibă nici o metodă abstractă moștenită fără implementare, fie să se declare ca fiind abstractă. Existența de metode neimplementate nu va permite instanțierea clasei respective.

5.6 Interfețe

O interfață definește un contract. O clasă sau o structură care implementează o interfață aderă la acest contract. Relația dintre o interfață și un tip care o implementează este deosebită de cea existentă între clase (este un/o): este o relație de implementare.

O interfață conține metode, proprietăți, evenimente, indexatori. Ea însă nu va conține implementări pentru acestea, doar declarații. Declarația unei interfețe se face astfel:

atribute_{opt} modificali-de-interfață_{opt} **interface** identificator baza-interfeței_{opt} corp-interfață ;_{opt}

Modificali de interfață sunt: *new*, *public*, *protected*, *internal*, *private*. O interfață poate să moștenească de la zero sau mai multe interfețe. Corpul interfeței conține declarații de metode, fără implementări. Orice metodă are gradul de acces public. Nu se poate specifica pentru o metodă din interiorul unei interfețe: *abstract*, *public*, *protected*, *internal*, *private*, *virtual*, *override*, ori *static*.

Exemplu:

```
interface IStorable
{
    void Read( );
    void Write();
}
```

O clasă care implementează o astfel de interfață se declară ca mai jos:

```
class Document: IStorable
{
    public void Read(){/*cod*/}
    public void Write(){/*cod*/}
    //alte declaratii
}
```

O clasă care implementează o interfață trebuie să definească toate metodele care se regăsesc în interfața respectivă. Următoarele reguli trebuie respectate la implementarea de interfețe:

1. Tipul de retur al metodei din clasă trebuie să coincidă cu tipul de retur al metodei din interfață
2. Tipul parametrilor formali din metodă trebuie să fie același cu tipul parametrilor formali din interfață

3. Metoda din clasă trebuie să fie declarată publică și nestatică.

Aceste implementări pot fi declarate folosind specificatorul *virtual* (deci sub-clasele clasei curente pot folosi *new* și *override*).

Exemplu:

```
using System;
interface ISavable
{
    void Read();
    void Write();
}
public class TextFile : ISavable
{
    public virtual void Read()
    {
        Console.WriteLine("TextFile.Read()");
    }
    public void Write()
    {
        Console.WriteLine("TextFile.Write()");
    }
}
public class ZipFile : TextFile
{
    public override void Read()
    {
        Console.WriteLine("ZipFile.Read()");
    }
    public new void Write()
    {
        Console.WriteLine("ZipFile.Write()");
    }
}
public class Test
{
    static void Main()
    {
        Console.WriteLine("\nTextFile reference to ZipFile");
        TextFile textRef = new ZipFile();
        textRef.Read();
        textRef.Write();
    }
}
```

```

    Console.WriteLine("\nISavable reference to ZipFile");
    ISavable savableRef = textRef as ISavable;
    if(savableRef != null)
    {
        savableRef.Read();
        savableRef.Write();
    }
    Console.WriteLine("\nZipFile reference to ZipFile");
    ZipFile zipRef = textRef as ZipFile;
    if(zipRef!= null)
    {
        zipRef.Read();
        zipRef.Write();
    }
}
}

```

La ieșire se va afișa:

```

TextFile reference to ZipFile
ZipFile.Read()
TextFile.Write()

```

```

ISavable reference to ZipFile
ZipFile.Read()
TextFile.Write()

```

```

ZipFile reference to ZipFile
ZipFile.Read()
ZipFile.Write()

```

În exemplul de mai sus se folosește operatorul *as* pentru a obține o referință la interfețe, pe baza obiectelor create. În general, se preferă ca apelul metodelor care sunt implementate din interfață să se facă via o referință la interfața respectivă, obținută prin intermediul operatorului *as* (ca mai sus) sau după o testare prealabilă prin *is* urmată de conversie explicită, ca mai jos:

```

if (textRef is ISavable)
{
    ISavable is = (ISavable)textRef;
    is.Read();//etc
}

```

În general, dacă se dorește doar răspunsul la întrebarea ”este obiectul curent un implementator al interfeței *I*?”, atunci se recomandă folosirea operatorului *is*. Dacă se știe că va trebui făcută și o conversie la tipul interfață, atunci este mai eficientă folosirea lui *as*. Afirmatia se bazează pe studiul codului IL rezultat în fiecare caz.

Să presupunem că exista o interfață *I* având metoda *M()* care este implementată de o clasă *C*, care definește metoda *M()*. Este posibil ca această metodă să nu aibă o semnificație în afara clasei *C*, ca atare a e de dorit ca metoda *M()* să nu fie declarată publică. Mecanismul care permite acest lucru se numește *implementare explicită*. Această tehnică permite ascunderea metodelor moștenite dintr-o interfață, acestea devenind private (calificarea lor ca fiind publice este semnalată ca o eroare). Implementarea explicită se obține prin calificarea numelui de metodă cu numele interfeței:

```
interface IMyInterface
{
    void F();
}
class MyClass : IMyInterface
{
    void IMyInterface.F()
    {
        //...
    }
}
```

Metodele din interfețe care s-au implementat explicit nu pot fi declarate *abstract*, *virtual*, *override*, *new*. Mai mult, asemenea metode nu pot fi accesate direct prin intermediul unui obiect (*obiect.NumeMetoda*), ci doar prin intermediul unei conversii către interfață respectivă, deoarece prin implementare explicită a metodelor aceste devin private și singura modalitate de acces a lor este upcasting-ul către interfață.

Exemplu:

```
using System;
public interface IDataBound
{
    void Bind();
}

public class EditBox : IDataBound
{
```

```

// implementare de IDataBound
void IDataBound.Bind()
{
    Console.WriteLine("Binding to data store...");
}

class NameHidingApp
{
    public static void Main()
    {
        Console.WriteLine();
        EditBox edit = new EditBox();
        Console.WriteLine("Apel EditBox.Bind()...");
        //EROARE: Aceasta linie nu se compileaza deoarece metoda
        //Bind nu mai exista ca metoda publica in clasa EditBox
        edit.Bind();
        Console.WriteLine();

        IDataBound bound = (IDataBound)edit;
        Console.WriteLine("Apel (IDataBound) EditBox.Bind()...");
        // Functioneaza deoarece s-a facut conversie la IDataBound
        bound.Bind();
    }
}

```

Este posibil ca un tip să implementeze mai multe interfețe. Atunci când două interfețe au o metodă cu aceeași semnătură, programatorul are mai multe variante de lucru. Cel mai simplu, el poate să furnizeze o singură implementare pentru ambele metode, ca mai jos:

```

interface IFriendly
{
    void GreetOwner() ;
}
interface IAffectionate
{
    void GreetOwner() ;
}
abstract class Pet
{
    public virtual void Eat()

```

```
{
    Console.WriteLine( "Pet.Eat" ) ;
}
}
class Dog : Pet, IAffectionate, IFriendly
{
    public override void Eat()
    {
        Console.WriteLine( "Dog.Eat" ) ;
    }
    public void GreetOwner()
    {
        Console.WriteLine( "Woof!" ) ;
    }
}
```

O altă modalitate este să se specifice implicit care metodă este implementată.

```
class Dog : Pet, IAffectionate, IFriendly
{
    public override void Eat()
    {
        Console.WriteLine( "Dog.Eat" ) ;
    }
    void IAffectionate.GreetOwner()
    {
        Console.WriteLine( "Woof!" ) ;
    }
    void IFriendly.GreetOwner()
    {
        Console.WriteLine( "Jump up!" ) ;
    }
}
}
public class Pets
{
    static void Main()
    {
        IFriendly mansBestFriend = new Dog() ;
        mansBestFriend.GreetOwner() ;
        (mansBestFriend as IAffectionate).GreetOwner() ;
    }
}
```

La ieșire se va afișa:

```
Jump up!  
Woof!
```

Dacă însă în clasa Dog se adaugă metoda

```
public void GreetOwner()  
{  
    Console.WriteLine( "Woof!" ) ;  
}
```

(care a fost inițial definită), atunci se poate face apel de tipul `dog.GreetOwner()` (dog este instanță de Dog); apelurile de metode din interfață rămân de asemenea valide. Rezultatul este ca și la exemplul precedent: se afișează mesajul Woof.

5.6.1 Clase abstracte sau interfețe?

Atât interfețele cât și clasele abstracte au comportamente similare și pot fi folosite în situații similare. Dar totuși ele nu se pot substitui reciproc. Câteva principii generale de utilizare a lor sunt date mai jos.

Dacă o relație se poate exprima mai degrabă ca “este un/o” decât altfel, atunci entitatea de bază ar trebui gândită ca o clasă abstractă.

Un alt aspect este bazat pe obiectele care ar folosi capabilitățile din tipul de bază. Dacă aceste capabilități ar fi folosite de către obiecte care nu sunt legate între ele, atunci ar fi indicată o interfață.

Dezavantajul claselor abstracte este că nu poate fi decât bază unică pentru orice altă clasă.

5.7 Tipuri parțiale

Începând cu versiunea 2.0 a platformei .NET este posibil ca definiția unei clase, interfețe sau structuri să fie făcută în mai multe fișiere sursă. Definiția clasei se obține din reuniunea părților componente, lucru făcut automat de către compilator. Această spargere în fragmente este benefică în următoarele cazuri:

- atunci când se lucrează cu proiecte mari, este posibil ca la o clasă să trebuiască să lucreze mai mulți programatori simultan - fiecare concentrându-se pe aspecte diferite.

- când se lucrează cu cod generat automat, acesta poate fi scris separat astfel încât programatorul să nu interfereze accidental cu el. Situația este frecvent întâlnită în cazul aplicațiilor de tip Windows forms sau Web services.

De exemplu, pentru o formă nou creată (numită Form1) mediul Visual Studio va scrie un fișier numit Form1.Designer.cs care conține partea de inițializare a controalelor și componentelor introduse de utilizator. Partea de tratare a evenimentelor, constructori, etc este definită într-un alt fișier (Form1.cs).

Declararea unei părți a unei clase se face folosind cuvântul cheie *partial* înaintea lui *class*.

Exemplu:

```
//fișierul Browser1.cs
public partial class Browser
{
    public void OpenPage(String uri)
    {...}
}
//fișierul Browser2.cs
public partial class Browser
{
    public void DiscardPage(String uri)
    {...}
}
```

Următoarele sunt valabile pentru tipuri parțiale:

- cuvântul *partial* trebuie să apară exact înainte cuvintelor: *class*, *interface*, *struct*
- dacă pentru o parte se specifică un anumit grad de acces, aceasta nu trebuie să ducă la conflicte cu declarațiile din alte părți
- dacă o parte de clasă este declarată ca abstractă, atunci întreaga clasă este considerată abstractă
- dacă o parte declară clasa ca fiind *sealed*, atunci întreaga clasă este considerată sealed
- dacă o parte declară că moștenește o clasă, atunci într-o altă parte nu se mai poate specifica o altă derivare
- părți diferite pot să declare că se implementează interfețe multiple

- aceleași câmpuri și metode nu pot fi definite în mai multe părți.
- clasele interioare pot să fie declarate în părți diferite, chiar dacă clasa conținătoare e definită într-un singur fișier:

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

- Următoarele elemente vor fi renite pentru definiția clasei: comentarii XML, interfețe, attribute, membri.

Exemplu:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

este echivalent cu:

```
class Earth : Planet, IRotate, IRevolve { }
```

5.8 Structuri

Structurile reprezintă tipuri de date asemănătoare claselor, cu principala diferență că sunt tipuri valoare (o astfel de variabilă va conține direct valoarea, și nu o adresă de memorie). Sunt considerate versiuni “ușoare” ale claselor, sunt folosite predilect pentru tipuri pentru care aspectul comportamental este mai puțin pronunțat.

Declarația unei structuri se face astfel:

`attributeopt modificali-de-structopt struct identifiator :interfețeopt corp ;opt`
 Modificali de structură sunt: *new*, *public*, *protected*, *internal*, *private*. O structură este automat derivată din *System.ValueType*, care la rândul ei este derivată din *System.Object*; de asemenea, este automat considerată *sealed* (nederivabilă). Poate însă să implementeze una sau mai multe interfețe.

O structură poate să conțină declarații de constante, câmpuri, metode, proprietăți, evenimente, indexatori, operatori, constructori, constructori statici, tipuri interioare. Nu poate conține destructor.

La atribuire, se face o copiere a valorilor conținute de către sursă în destinație (indiferent de tipul câmpurilor: valoare sau referință).

Exemplu:

```
using System;
public struct Point
{
    public Point(int xCoordinate, int yCoordinate)
    {
        xVal = xCoordinate;
        yVal = yCoordinate;
    }
    public int X
    {
        get
        {
            return xVal;
        }
        set
        {
            xVal = value;
        }
    }
    public int Y
    {
        get
        {
            return yVal;
        }
        set
        {
            yVal = value;
        }
    }
}

public override string ToString( )
{
    return (String.Format("{0}, {1}", xVal,yVal));
}
```

```

    }
    public int xVal;
    public int yVal;
}

public class Tester
{
    public static void MyFunc(Point loc)
    {
        loc.X = 50;
        loc.Y = 100;
        Console.WriteLine("In MyFunc loc: {0}", loc);
    }
    static void Main( )
    {
        Point loc1 = new Point(200,300);
        Console.WriteLine("Loc1 location: {0}", loc1);
        MyFunc(loc1);
        Console.WriteLine("Loc1 location: {0}", loc1);
    }
}

```

După cum este dat în exemplul de mai sus, crearea unei instanțe se face folosind operatorul *new*; dar în acest caz, nu se va crea o instanță în memoria heap, ci pe stivă. Transmiterea lui *loc1* ca parametru se face prin valoare, adică metoda *myFunc* nu face decât să modifice o copie de pe stivă a lui *loc1*. La revenire, se va afișa tot valoarea originală, deoarece *loc1* a rămas nemodificat:

```

Loc1 location: 200, 300
In MyFunc loc: 50, 100
Loc1 location: 200, 300

```

Deseori pentru o structură se declară câmpurile ca fiind publice, pentru a nu mai fi necesare definirea accesoriilor (simplificare implementării). Alți programatori consideră însă că accesarea membrilor trebuie să se facă precum la clase, folosind proprietăți. Oricare ar fi alegerea, limbajul o sprijină.

Alte aspecte demne de reținut:

- Câmpurile nu pot fi inițializate la declarare; altfel spus, dacă în exemplul de mai sus se scria:

```
public int xVal = 10;
public int yVal = 20;
```

s-ar fi semnalat o eroare la compilare.

- Nu se poate defini un constructor implicit. Cu toate acestea, compilatorul va crea un astfel de constructor, care va inițializa câmpurile la valorile lor implicite (0 pentru tipuri numerice sau pentru enumerări, *false* pentru *bool*, *null* pentru tipuri referință).

Pentru tipul *Point* de mai sus, următoarea secvență de cod este corectă:

```
Point a = new Point(0, 0);
Point b = new Point();
```

și duce la crearea a două puncte cu abscisele și ordonatele 0. Un constructor implicit este apelat atunci când se creează un tablou de structuri:

```
Point[] points = new Points[10];
for( int i=0; i<points.Length; i++ )
{
    Console.WriteLine(points[i]);
}
```

va afișa de 10 ori puncte de coordonate (0, 0). De asemenea este apelat la inițializarea membrilor unei clase (care se face înainte de orice constructor).

De menționat pentru exemplul anterior că se creează un obiect de tip tablou în heap, după care în interiorul lui (și nu pe stivă!) se creează cele 10 puncte (alocare inline).

- Nu se poate declara destructor. Aceștia se declară numai pentru clase.
- Dacă programatorul definește un constructor, atunci acesta trebuie să dea valori inițiale pentru câmpurile conținute, altfel apare eroare la compilare.
- Dacă pentru instanțierea unei structuri declarate în interiorul unei metode sau bloc de instrucțiuni în general nu se apelează *new*, atunci respectiva instanță nu va avea asociată nici o valoare (constructorul implicit nu este apelat automat!). Nu se poate folosi respectiva variabilă de tip structură decât după ce i se inițializează toate câmpurile:

```
Point p;
Console.WriteLine(p);
```

va duce la apariția erorii de compilare:

```
Use of unassigned local variable 'p'
```

Dar după niște asignări de tipul:

```
p.xVal=p.yVal=0;
```

afișarea este posibilă (practic, orice apel de metodă pe instanță este acum acceptat).

- Dacă se înțelege definirea unei structuri care conține un câmp de tipul structurii, atunci va apărea o eroare de compilare:

```
struct MyStruct
{
    MyStruct s;
}
```

va genera un mesaj din partea compilatorului:

```
Struct member 'MyStruct.s' of type 'MyStruct' causes a
cycle in the structure layout
```

- Dacă o instanță este folosită acolo unde un *object* este necesar, atunci se va face automat o conversie implicită către *System.Object* (boxing). Ca atare, utilizarea unei structuri poate duce (dar nu obligatoriu, ci în funcție de context) la un overhead datorat conversiei.

5.8.1 Structuri sau clase?

Structurile pot fi mult mai eficiente în alocarea memoriei atunci când sunt reținute într-un tablou. De exemplu, crearea unui tablou de 100 de elemente de tip *Point* (de mai sus) va duce la crearea unui singur obiect (tabloul), iar cele 100 de instanțe de tip structură ar fi alocate inline în vectorul creat (și nu referințe ale acestora). Dacă *Point* ar fi declarat ca și clasă, ar fi fost necesară crearea a 101 instanțe de obiecte în heap (un obiect pentru tablou, alte 100 pentru puncte), ceea ce ar duce la mai mult lucru pentru garbage collector.

Dar în cazul în care structurile sunt folosite în colecții de tip `Object` (de exemplu un `ArrayList`), se va face automat un boxing, ceea ce duce la overhead (memorie și timp = resurse suplimentare). De asemenea, la transmiterea prin valoare a unei structuri, se va face copierea tuturor câmpurilor conținute pe stivă, ceea ce poate duce la un overhead semnificativ.

Curs 6

Delegați. Evenimente. Tratarea excepțiilor

6.1 Tipul delegat

În programare deseori apare următoarea situație: trebuie să se execute o anumită acțiune, dar nu se știe de dinainte care anume, sau chiar ce obiect va trebui efectiv utilizat. De exemplu, un buton poate ști că trebuie să anunțe pe oricine este interesat despre faptul că fost apăsat, dar nu va ști aprioric cum va fi tratat acest eveniment. Mai degrabă decât să se lege butonul de un obiect particular, butonul va declara un delegat, pentru care clasa interesată de evenimentul de apăsare va da o implementare.

Fiecare acțiune pe care utilizatorul o execută pe o interfață grafică declanșează un eveniment. Alte evenimente se pot declanșa independent de acțiunile utilizatorului: sosirea unui email, terminarea copierii unor fișiere, sfârșitul unei interogări pe o bază de date, etc. Un eveniment este o încapsulare a ideii că “se întâmplă ceva” la care programul trebuie să răspundă. Evenimentele și delegații sunt strâns legate deoarece răspunsul la acest eveniment se va face de către un event handler, care este legat de eveniment printr-un delegat,

Un delegat este un tip referință folosit pentru a încapsula o metodă cu un anumit antet (tipul parametrilor formali și tipul de retur). Orice metodă care are acest antet poate fi legată la un anumit delegat. Într-un limbaj precum C++, acest lucru se rezolvă prin intermediul pointerilor la funcții. Delegații rezolvă aceeași problemă, dar într-o manieră orientată obiect și cu garanții asupra siguranței codului rezultat, precum și cu o ușoară generalizare (vezi delegații multicast).

Un delegat este creat după următoarea sintaxă:

atribute_{opt} modificatori-de-delegat_{opt} delegate tip-retur identificator(lista-param-formali_{opt});

Modificatorul de delegat poate fi: *new*, *public*, *protected*, *internal*, *private*. Un delegat se poate specifica atât în interiorul unei clase, cât și în exteriorul ei, fiind de fapt o declarație de clasă derivată din *System.Delegate*. Dacă este declarat în interiorul unei clase, atunci este și static (a se vedea statutul claselor interioare).

Exemplu:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

6.1.1 Utilizarea delegaților pentru a specifica metode la runtime

Să presupunem că se dorește crearea unei clase container simplu numit *Pair* care va conține două obiecte pentru care va putea face și sortare. Nu se va ști aprioric care va fi tipul obiectelor conținute, deci se va folosi pentru ele tipul *object*. Dar sortarea celor două obiecte se va face diferit, în funcție de tipul lor efectiv: de exemplu pentru niște persoane (clasa *Student* în cele ce urmează) se va face după nume, pe când pentru animale (clasa *Dog*) se va face după alt criteriu: greutatea. Containerul *Pair* va trebui să facă față acestor clase diferite. Rezolvarea se va da prin delegați.

Clasa *Pair* va defini un delegat, *WhichIsFirst*. Metoda *Sort* de ordonare va prelua ca (unic) parametru o instanță a metodei *WhichIsFirst*, care va implementa relația de ordine, în funcție de tipul obiectelor conținute. Rezultatul unei comparații între două obiecte va fi de tipul enumerare *Comparison*, definit de utilizator:

```
public enum Comparison
{
    theFirstComesFirst = 0,
    //primul obiect din colectie este primul in ordinea sortarii
    theSecondComesFirst = 1
    //al doilea obiect din colectie este primul in ordinea sortarii
}
```

Delegatul (tipul de metodă care realizează compararea) se declară astfel:

```
//declarare de delegat
public delegate Comparison WhichIsFirst(
    object obj1, object obj2);
```

Clasa *Pair* se declară după cum urmează:

```

public class Pair
{
    //tabloul care contine cele doua obiecte
    private object[] thePair = new object[2];

    //constructorul primeste cele doua obiecte continute
    public Pair( object firstObject, object secondObject)
    {
        thePair[0] = firstObject;
        thePair[1] = secondObject;
    }

    //metoda publica pentru ordonarea celor doua obiecte
    //dupa orice criteriu
    public void Sort( WhichIsFirst theDelegatedFunc )
    {
        if (theDelegatedFunc(thePair[0],thePair[1]) ==
            Comparison.theSecondComesFirst)
        {
            object temp = thePair[0];
            thePair[0] = thePair[1];
            thePair[1] = temp;
        }
    }
    //metoda ce permite tiparirea perechii curente
    //se foloseste de polimorfism - vezi mai jos
    public override string ToString( )
    {
        return thePair[0].ToString()+" , "+thePair[1].ToString();
    }
}

```

Clasele Student și Dog sunt:

```

public class Dog
{
    public Dog(int weight)
    {
        this.weight=weight;
    }
    //Ordinea este data de greutate
    public static Comparison WhichDogComesFirst(

```

```

        Object o1, Object o2)
    {
        Dog d1 = o1 as Dog;
        Dog d2 = o2 as Dog;
        return d1.weight > d2.weight ?
            Comparison.theSecondComesFirst :
            Comparison.theFirstComesFirst;
    }
    //pentru afisarea greutatii unui caine
    public override string ToString( )
    {
        return weight.ToString( );
    }
    private int weight;
}

public class Student
{
    public Student(string name)
    {
        this.name = name;
    }
    //studentii sunt ordonati alfabetice
    public static Comparison WhichStudentComesFirst(
        Object o1, Object o2)
    {
        Student s1 = o1 as Student;
        Student s2 = o2 as Student;
        return (String.Compare(s1.name, s2.name) < 0 ?
            Comparison.theFirstComesFirst :
            Comparison.theSecondComesFirst);
    }
    //pentru afisarea numelui unui student
    public override string ToString( )
    {
        return name;
    }
    private string name;
}

```

Clasa de test este:

```

public class Test
{
    public static void Main( )
    {
        //creaza cate doua obiecte
        //de tip Student si Dog
        //si containerii corespunzatori
        Student Stacey = new Student(“Stacey”);
        Student Jesse = new Student ( “Jess”);
        Dog Milo = new Dog(10);
        Dog Fred = new Dog(5);
        Pair studentPair = new Pair(Stacey, Jesse);
        Pair dogPair = new Pair(Milo, Fred);
        Console.WriteLine(“studentPair\t: {0}”, studentPair);
        Console.WriteLine(“dogPair\t: {0}”, dogPair);
        //Instantiaza delegatii
        WhichIsFirst theStudentDelegate =
            new WhichIsFirst(Student.WhichStudentComesFirst);
        WhichIsFirst theDogDelegate =
            new WhichIsFirst(Dog.WhichDogComesFirst);
        //sortare folosind delegatii
        studentPair.Sort(theStudentDelegate);
        Console.WriteLine(“Dupa sortarea pe studentPair\t: {0}”,
            studentPair.ToString( ));
        dogPair.Sort(theDogDelegate);
        Console.WriteLine(“Dupa sortarea pe dogPair\t\t: {0}”,
            dogPair.ToString( ));
    }
}

```

6.1.2 Delegații statice

Unul din aspectele neelegante ale exemplului anterior este că e necesar ca în clasa *Test* să se instanțieze delegații care sunt necesari pentru a ordona obiectele din *Pair*. O modalitate mai bună este să se obțină delegații direct din clasele *Student* și *Dog*. Acest lucru se obține prin crearea unui delegat static în interiorul fiecărei clase:

```

public static readonly WhichIsFirst OrderStudents =
    new WhichIsFirst(Student.WhichStudentComesFirst);

```

(analog pentru clasa *Dog*; de remarcat că “static readonly” nu se poate înlocui cu “const”, deoarece inițializatorul nu este considerat expresie constantă). Declarația de mai sus se folosește astfel:

```
...
studentpair.Sort(Student.OrderStudent);
...
```

rezultatul fiind identic.

În [1] este dată și o implementare de delegat ca proprietate statică, aceasta ducând la crearea unui delegat doar în cazul în care este nevoie de el¹.

6.1.3 Multicasting

Uneori este nevoie ca un delegat să poată apela mai mult de o singură metodă. De exemplu, atunci când un buton este apăsat, se poate să vrei să efectuezi mai mult de o singură acțiune: să scrii într-un textbox un șir de caractere și să înregistrezi într-un fișier faptul că s-a apăsat acel buton (logging). Acest lucru s-ar putea rezolva prin construirea unui vector de delegați care să conțină toate metodele dorite, însă s-ar ajunge la un cod greu de urmărit și inflexibil; pentru un astfel de exemplu, a se vedea [1]. Mult mai simplu ar fi dacă unui delegat i s-ar putea atribui mai multe metode. Acest lucru se numește *multicasting* și este folosit intens la tratarea evenimentelor (vezi secțiunea următoare).

Orice delegat care returnează *void* este un delegat multicast, care poate fi tratat și ca un delegat single-cast. Doi delegați multicast pot fi combinați folosind semnul +. Rezultatul unei astfel de “adunări” este un nou delegat multicast care la apelare va invoca metodele conținute, în ordinea în care s-a făcut adunarea. De exemplu, dacă *Writer* și *Logger* sunt delegați care returnează *void*, atunci următoarea linie va produce combinarea lor într-un singur delegat:

```
myMulticastDelegate = Writer + Logger;
```

Se pot adăuga delegați multicast folosind operatorul +=, care va adăuga delegatul de la dreapta operatorului la delegatul multicast aflat în stânga sa:

```
myMulticastDelegate += Transmitter;
```

presupunând că *Transmitter* este compatibil cu *myMulticastDelegate* (are aceeași semnătură). Operatorul − = funcționează invers față de + =.

Exemplu:

¹Tehnică numită inițializaer târzie (lazy initialization)

```
using System;
//declaratia de delegat multicast
public delegate void StringDelegate(string s);

public class MyImplementingClass
{
    public static void WriteString(string s)
    {
        Console.WriteLine("Writing string {0}", s);
    }
    public static void LogString(string s)
    {
        Console.WriteLine("Logging string {0}", s);
    }
    public static void TransmitString(string s)
    {
        Console.WriteLine("Transmitting string {0}", s);
    }
}

public class Test
{
    public static void Main( )
    {
        //defineste trei obiecte delegat
        StringDelegate Writer,
            Logger, Transmitter;
        //defineste alt delegat
        //care va actiona ca un delegat multicast
        StringDelegate myMulticastDelegate;
        //Instantiaza primii trei delegati
        //dand metodele ce se vor incapsula
        Writer = new StringDelegate(
            MyImplementingClass.WriteString);
        Logger = new StringDelegate(
            MyImplementingClass.LogString);
        Transmitter = new StringDelegate(
            MyImplementingClass.TransmitString);
        //Invoca metoda delegat Writer
        Writer("String passed to Writer\n");
        //Invoca metoda delegat Logger
```

```

    Logger("String passed to Logger\n");
    //Invoca metoda delegat Transmitter
    Transmitter("String passed to Transmitter\n");
    //anunta utilizatorul ca va combina doi delegati
    Console.WriteLine(
        "myMulticastDelegate = Writer + Logger");
    //combina doi delegati, rezultatul este
    //assignat lui myMulticastDelagate
    myMulticastDelegate = Writer + Logger;
    //apelaeaza myMulticastDelegate
    //de fapt vor fi chemate cele doua metode
    myMulticastDelegate(
        "First string passed to Collector");
    //Anunta utilizatorul ca se va adauga al treilea delegat
    Console.WriteLine(
        "\nmyMulticastDelegate += Transmitter");
    //adauga al treilea delegat
    myMulticastDelegate += Transmitter;
    //invoca cele trei metode delagate
    myMulticastDelegate(
        "Second string passed to Collector");
    //anunta utilizatorul ca se va scoate delegatul Logger
    Console.WriteLine(
        "\nmyMulticastDelegate -= Logger");
    //scoate delegatul Logger
    myMulticastDelegate -= Logger;
    //invoca cele doua metode delegat ramase
    myMulticastDelegate(
        "Third string passed to Collector");
}
}

```

La ieșire vom avea:

```

Writing string String passed to Writer
Logging string String passed to Logger
Transmitting string String passed to Transmitter
myMulticastDelegate = Writer + Logger
Writing string First string passed to Collector
Logging string First string passed to Collector
myMulticastDelegate += Transmitter
Writing string Second string passed to Collector

```



```
Logging string Second string passed to Collector
Transmitting string Second string passed to Collector
myMulticastDelegate -= Logger
Writing string Third string passed to Collector
Transmitting string Third string passed to Collector
```

6.2 Evenimente

Interfețele grafice actuale cer ca un anumit program să răspundă la *evenimente*. Un eveniment poate fi de exemplu apăsarea unui buton, terminarea transferului unui fișier, selectarea unui meniu, etc; pe scurt, se întâmplă ceva la care trebuie să se dea un răspuns. Nu se poate prezice ordinea în care se petrec evenimentele, iar la apariția unuia se va cere reacționarea din partea sistemului soft.

Alte clase pot fi interesate în a răspunde la aceste evenimente. Modul în care vor reacționa va fi extrem de particular, iar obiectul care semnalează evenimentul (ex: clasa Button, la apăsarea unui buton) nu trebuie să știe modul în care se va răspunde. Butonul va comunica faptul că a fost apăsător, iar clasele interesate în acest eveniment vor reacționa în consecință.

6.2.1 Publicarea și subscrierea

În C#, orice obiect poate să *publice* un set de evenimente la care alte clase pot să *subscrie*. Când obiectul care a publicat evenimentul îl și semnalează, toate obiectele care au subscris la acest eveniment sunt notificate. În acest mod se definește o dependență de tip *one-to-many* între obiecte astfel încât dacă un obiect își schimbă starea, atunci toate celelalte obiecte dependente sunt notificate și modificate automat.

De exemplu, un buton poate să notifice un număr oarecare de observatori atunci când a fost apăsător. Butonul va fi numit *publicator*² deoarece publică evenimentul *Click* iar celelalte clase sunt numite *abonați*³ deoarece ele subscriu la evenimentul Click.

6.2.2 Evenimente și delegați

Tratarea evenimentelor în C# se face folosind delegați. Clasa ce publică definește un delegat pe care clasele abonate trebuie să îl implementeze. Când evenimentul este declanșat, metodele claselor abonate vor fi apelate prin

²Engl: publisher

³Engl: subscribers

intermediul delegatului (pentru care se prevede posibilitatea de a fi multicast, astfel încât să se permită mai mulți abonați).

Metodele care răspund la un eveniment se numesc *event handlers*. Prin convenție, un event handler în .NET Framework returnează *void* și preia doi parametri: primul parametru este sursa evenimentului (obiectul publicator); al doilea parametru are tip *EventArgs* sau derivat din acesta.

Declararea unui eveniment se face astfel:

atribute_{opt} modificali-de-eveniment_{opt} **event** tip nume—eveniment
Modificali-de-eveniment poate fi *abstract*, *new*, *public*, *protected*, *internal*, *private*, *static*, *virtual*, *sealed*, *override*, *extern*. *Tip* este un handler de eveniment (delegat multicast).

Exemplu:

```
public event SecondChangeHandler OnSecondChange;
```

Vom da mai jos un exemplu care va construi următoarele: o clasă *Clock* care folosește un eveniment (*OnSecondChange*) pentru a notifica potențialii abonați atunci când timpul local se schimbă cu o secundă. Tipul acestui eveniment este un delegat *SecondChangeHandler* care se declară astfel:

```
public delegate void SecondChangeHandler(  
    object clock, TimeInfoEventArgs timeInformation );
```

în conformitate cu metodologia de declarare a unui event handler, pomenită mai sus. Tipul *TimeInfoEventArgs* este definit de noi ca o clasă derivată din *EventArgs*:

```
public class TimeInfoEventArgs : EventArgs  
{  
    public TimeInfoEventArgs( int hour, int minute, int second )  
    {  
        this.hour = hour;  
        this.minute = minute;  
        this.second = second;  
    }  
    public readonly int hour;  
    public readonly int minute;  
    public readonly int second;  
}
```

Această clasă va conține informație despre timpul curent. Informația este accesibilă *readonly*.

Clasa *Clock* va conține o metodă *Run()*:

```
public void Run()
{
    for(;;)
    {
        //dormi 10 milisecunde
        Thread.Sleep(10);
        //obține timpul curent
        System.DateTime dt = System.DateTime.Now();
        //daca timpul s-a schimbat cu o secunda
        //atunci notifica abonatii
        if( dt.Second != second)
        //second este camp al clasei Clock
        {
            //creeaza obiect TimeInfoEventArgs
            //ce va fi transmis abonatilor
            TimeInfoEventArgs timeInformation =
                new TimeInfoEventArgs(dt.Hour, dt.Minute, dt.Second);
            //daca cineva este abonat, atunci anunta-l
            if (OnSecondChange != null)
            {
                OnSeconChange(this, timeInformation);
            }
        }
        //modifica timpul curent in obiectul Clock
        this.second = dt.Second;
        this.minute = dt.Minute;
        this.hour = dt.Hour;
    }
}
```

Metoda *Run* creează un ciclu infinit care interoghează periodic ceasul sistem. Dacă timpul s-a schimbat cu o secundă față de timpul precedent, se vor notifica toate obiectele abonate după care își va modifica starea, prin cele trei atribuiri finale.

Tot ce rămâne de făcut este să se scrie niște clase care să subscrie la evenimentul publicat de clasa *Clock*. Vor fi două clase: una numită *DisplayClock* care va afișa pe ecran timpul curent și o alta numită *LogCurrentTime* care ar trebui să înregistreze evenimentul într-un fișier, dar pentru simplitate va afișa doar la dispozitivul curent de ieșire informația tranmsisă:

```
public class DisplayClock
{
```

```

public void Subscribe(Clock theClock)
{
    theClock.OnSecondChange +=
        new Clock.SecondChangeHandler(TimeHasChanged);
}
void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
{
    Console.WriteLine("Current Time: {0}:{1}:{2}",
        ti.hour.ToString( ),
        ti.minute.ToString( ),
        ti.second.ToString( ));
}
}
public class LogCurrentTime
{
    public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(WriteLogEntry);
    }
    //Aceasta metoda ar trebui sa scrie intr-un fisier
    //dar noi vom scrie la consola
    void WriteLogEntry(
        object theClock, TimeInfoEventArgs ti)
    {
        Console.WriteLine("Logging to file: {0}:{1}:{2}",
            ti.hour.ToString( ),
            ti.minute.ToString( ),
            ti.second.ToString( ));
    }
}
}

```

De remarcat faptul că evenimentele sunt adăugate folosind operatorul +=.

Exemplul în întregime este dat mai jos:

```

using System;
using System.Threading;
//o clasa care va contine informatie despre eveniment
//in acest caz va contine informatie disponibila in clasa Clock
public class TimeInfoEventArgs : EventArgs
{

```

```
public TimeInfoEventArgs(int hour, int minute, int second)
{
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
public readonly int hour;
public readonly int minute;
public readonly int second;
}
//clasa care publica un eveniment: OnSecondChange
//clasele care se aboneaza vor subscrie la acest eveniment
public class Clock
{
    //delegatul pe care abonatii trebuie sa il implementeze
    public delegate void SecondChangeHandler(
        object clock, TimeInfoEventArgs timeInformation );
    //evenimentul ce se publica
    public event SecondChangeHandler OnSecondChange;
    //ceasul este pornit si merge la infinit
    //va declansa un eveniment pentru fiecare secunda trecuta
    public void Run( )
    {
        for(;;)
        {
            //inactiv 10 ms
            Thread.Sleep(10);
            //citeste timpul curent al sistemului
            System.DateTime dt = System.DateTime.Now;
            //daca s-a schimbat fata de secunda anterior inregistrata
            //atunci notifica pe abonati
            if (dt.Second != second)
            {
                //creaza obiectul TimeInfoEventArgs
                //care va fi transmis fiecarui abonat
                TimeInfoEventArgs timeInformation =
                    new TimeInfoEventArgs(
                        dt.Hour,dt.Minute,dt.Second);
                //daca cineva a scris la acest eveniment
                //atunci anunta-l
                if (OnSecondChange != null)
```

```

        {
            OnSecondChange(this,timeInformation);
        }
    }
    //modifica starea curenta
    this.second = dt.Second;
    this.minute = dt.Minute;
    this.hour = dt.Hour;
}
}
private int hour;
private int minute;
private int second;
}
//un observator (abonat)
//DisplayClock va subscrie la evenimentul lui Clock
//DisplayClock va afisa timpul curent
public class DisplayClock
{
    //dandu-se un obiect clock, va subscrie
    //la evenimentul acestuia
    public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(TimeHasChanged);
    }
    //handlerul de eveniment de pe partea
    //clasei DisplayClock
    void TimeHasChanged(
        object theClock, TimeInfoEventArgs ti)
    {
        Console.WriteLine("Current Time: {0}:{1}:{2}",
            ti.hour.ToString( ),
            ti.minute.ToString( ),
            ti.second.ToString( ));
    }
}
//un al doilea abonat care ar trebui sa scrie intr-un fisier
public class LogCurrentTime
{
    public void Subscribe(Clock theClock)

```

```
{
    theClock.OnSecondChange +=
        new Clock.SecondChangeHandler(WriteLogEntry);
}
//acest handler ar trebui sa scrie intr-un fisier
//dar va scrie la standard output
void WriteLogEntry(
    object theClock, TimeInfoEventArgs ti)
{
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
        ti.hour.ToString( ),
        ti.minute.ToString( ),
        ti.second.ToString( ));
}
}
public class Test
{
    static void Main( )
    {
        //creaza un obiect de tip Clock
        Clock theClock = new Clock( );
        //creaza un obiect DisplayClock care
        //va subscrie la evenimentul obiectului
        //Clock anterior creat
        DisplayClock dc = new DisplayClock( );
        dc.Subscribe(theClock);
        //analog se creeaza un obiect de tip LogCurrentTime
        //care va subscrie la acelasi eveniment
        //ca si obiectul DisplayClock
        LogCurrentTime lct = new LogCurrentTime( );
        lct.Subscribe(theClock);
        //porneste ceasul
        theClock.Run( );
    }
}
```

La ieşire se va afişa:

```
Current Time: 14:53:56
Logging to file: 14:53:56
Current Time: 14:53:57
Logging to file: 14:53:57
```

Current Time: 14:53:58

Logging to file: 14:53:58

Current Time: 14:53:59

Logging to file: 14:53:59

6.2.3 Comentarii

S-ar putea pune următoarea întrebare: de ce este nevoie de o astfel de redirectare de eveniment, când în metoda *Run()* se poate afișa direct pe ecran sau într-un fișier informația cerută? Avantajul abordării anterioare este că se pot crea oricâte clase care să fie notificate atunci când acest eveniment se declanșează. Clasele abonate nu trebuie să știe despre modul în care lucrează clasa *Clock*, iar clasa *Clock* nu trebuie să știe despre clasele care vor subscrie la evenimentul său. Similar, un buton poate să publice un eveniment *OnClick* și orice număr de obiecte pot subscrie la acest eveniment, primind o notificare atunci când butonul este apăsat.

Publicatorul și abonații sunt decuplați. Clasa *Clock* poate să modifice modalitatea de detectare a schimbării de timp fără ca acest lucru să impună o schimbare în clasele abonate. De asemenea, clasele abonate pot să își modifice modul de tratare a evenimentului, în mod transparent față de clasa *Clock*. Toate aceste caracteristici fac întreținerea codului extrem de facilă.

6.3 Tratarea excepțiilor

C#, la fel ca alte limbaje, permite tratarea erorilor și a situațiilor deosebite prin excepții. O excepție este un obiect care încapsulează informație despre o situație anormală. Ea este folosită pentru a semnaliza contextul în care apare situația deosebită

Un programator nu trebuie să confunde tratarea excepțiilor cu erorile sau bug-urile. Un bug este o eroare de programare care ar trebui să fie fixată înainte de livrarea codului. Excepțiile nu sunt gândite pentru a preveni bug-urile (cu toate că un bug poate să ducă la apariția unei excepții), pentru că acestea din urmă ar trebui să fie eliminate.

Chiar dacă se scot toate bug-urile, vor exista erori predictibile dar neprevenibile, precum deschiderea unui fișier al cărui nume este greșit sau împărțiri la 0. Nu se pot preveni astfel de situații, dar se pot manipula astfel încât nu vor duce la prăbușirea programului. Când o metodă întâlnește o situație excepțională, atunci se va arunca o excepție; cineva va trebui să sesizeze (să “prindă”) această excepție, sau eventual să lase o funcție de nivel superior să

o trateze. Dacă nimeni nu tratează această excepție, atunci CLR o va face, dar aceasta duce la oprirea *thread-ului*⁴.

6.3.1 Tipul *Exception*

În C# se pot arunca ca excepții obiecte de tip *System.Exception* sau derivate ale acestuia. Există o ierarhie de excepții care se pot folosi, sau se pot crea propriile tipuri excepție.

Enumerăm următoarele metode și proprietăți relevante ale clasei *Exception*:

- *public Exception()*, *public Exception(string)*, *public Exception(string, Exception)* - constructori; ultimul preia un obiect de tip *Exception* (sau de tip clasă derivată) care va fi încapsulat în instanța curentă; o excepție poate deci să conțină în interiorul său o instanță a altei excepții (cea care a fost de fapt semnalată inițial).
- *public virtual string HelpLink {get; set;}* obține sau setează un link către un fișier help asociat acestei excepții; poate fi de asemenea o adresa Web (URL)
- *public Exception InnerException {get;}* returnează excepția care este încorporată în excepția curentă
- *public virtual string Message {get;}* obține un mesaj care descrie excepția curentă
- *public virtual string Source {get; set;}* obține sau setează numele aplicației sau al obiectului care a cauzat eroarea
- *public virtual string StackTrace {get;}* obține o reprezentare string a apelurilor de metode care au dus la apariția acestei excepții
- *public MethodBase TargetSite {get;}* obține metoda care a aruncat excepția curentă⁵

6.3.2 Aruncarea și prinderea excepțiilor

Aruncarea cu *throw*

Aruncarea unei excepții se face folosind instrucțiunea *throw*. Exemplu:

⁴Și nu neapărat a întregului proces!

⁵*MethodBase* este o clasă care pune la dispoziție informații despre metodele și constructorii unei clase

```
throw new System.Exception();
```

Aruncarea unei excepții oprește execuția metodei curente, după care CLR începe să caute un manipulator de excepție. Dacă un handler de excepție nu este găsit în metoda curentă, atunci CLR va curăța stiva, ajungându-se la metoda apelantă. Fie undeva în lanțul de metode care au fost apelate se găsește un exception handler, fie thread-ul curent este terminat de către CLR.

Exemplu:

```
using System;
public class Test
{
    public static void Main( )
    {
        Console.WriteLine("Enter Main...");
        Test t = new Test( );
        t.Func1( );
        Console.WriteLine("Exit Main...");
    }
    public void Func1( )
    {
        Console.WriteLine("Enter Func1...");
        Func2( );
        Console.WriteLine("Exit Func1...");
    }
    public void Func2( )
    {
        Console.WriteLine("Enter Func2...");
        throw new System.Exception( );
        Console.WriteLine("Exit Func2...");
    }
}
```

Se exemplifică apelul de metode: *Main()* apelează *Func1()*, care apelează *Func2()*; aceasta va arunca o excepție. Deoarece lipsește un event handler care să trateze această excepție, se va întrerupe thread-ul curent (și fiind singurul, și întregul proces) de către CLR, iar la ieșire vom avea:

```
Enter Main...
Enter Func1...
Enter Func2...
```

```
Exception occurred: System.Exception: An exception of type
System.Exception was thrown at Test.Func2( )
in ...Test.cs:line 24
at Test.Func1( )
in ...Test.cs:line 18
at Test.Main( )
in ...Test.cs:line 12
```

Deoarece este aruncată o excepție, în metoda *Func2()* nu se va mai executa ultima linie, ci CLR-ul va începe imediat căutarea event handler-ului care să trateze excepția. La fel, nu se execută nici ultima linie din *Func1()* sau din *Main()*.

Prinderea cu *catch*

Prinderea și tratarea excepției se poate face folosind un bloc *catch*, creat prin intermediul instrucțiunii *catch*.

Exemplu:

```
using System;
public class Test
{
    public static void Main( )
    {
        Console.WriteLine("Enter Main...");
        Test t = new Test( );
        t.Func1( );
        Console.WriteLine("Exit Main...");
    }
    public void Func1( )
    {
        Console.WriteLine("Enter Func1...");
        Func2( );
        Console.WriteLine("Exit Func1...");
    }
    public void Func2( )
    {
        Console.WriteLine("Enter Func2...");
        try
        {
            Console.WriteLine("Entering try block...");
            throw new System.Exception( );
        }
    }
}
```

```

        Console.WriteLine('Exiting try block...');
    }
    catch
    {
        Console.WriteLine('Exception caught and handled.');
```

Se observă că s-a folosit un bloc *try* pentru a delimita instrucțiunile care vor duce la apariția excepției. În momentul în care se aruncă excepția, restul instrucțiunilor din blocul *try* se ignoră și controlul este preluat de către blocul *catch*. Deoarece excepția a fost tratată, CLR-ul nu va mai opri procesul. La ieșire se va afișa:

```

Enter Main...
Enter Func1...
Enter Func2...
Entering try block...
Exception caught and handled.
Exit Func2...
Exit Func1...
Exit Main...
```

Se observă că în blocul *catch* nu s-a specificat tipul de excepție care se prinde; asta înseamnă că se va prinde orice excepție se va arunca, indiferent de tipul ei. Chiar dacă excepția este tratată, execuția nu se va relua de la instrucțiunea care a produs excepția, ci se continuă cu instrucțiunea de după blocul *catch*.

Uneori, prinderea și tratatarea excepției nu se poate face în funcția apelată, ci doar în funcția apelantă. Exemplu:

```

using System;
public class Test
{
    public static void Main( )
    {
        Console.WriteLine('Enter Main...');
        Test t = new Test( );
        t.Func1( );
        Console.WriteLine('Exit Main...');
```

```
}  
public void Func1( )  
{  
    Console.WriteLine(''Enter Func1...'');  
    try  
    {  
        Console.WriteLine(''Entering try block...'');  
        Func2( );  
        Console.WriteLine(''Exiting try block...'');  
    }  
    catch  
    {  
        Console.WriteLine(''Exception caught and handled.'');  
    }  
    Console.WriteLine(''Exit Func1...'');  
}  
public void Func2( )  
{  
    Console.WriteLine(''Enter Func2...'');  
    throw new System.Exception( );  
    Console.WriteLine(''Exit Func2...'');  
}  
}
```

La ieșire se va afișa:

```
Enter Main...  
Enter Func1...  
Entering try block...  
Enter Func2...  
Exception caught and handled.  
Exit Func1...  
Exit Main...
```

Este posibil ca într-o secvență de instrucțiuni să se arunce mai multe tipuri de excepții, în funcție de natura stării apărute. În acest caz, prinderea excepției printr-un bloc *catch* generic, ca mai sus, nu este utilă; am vrea ca în funcție de natura excepției aruncate, să facem o tratare anume. Se sugerează chiar să nu se folosească această construcție de prindere generică, deoarece în majoritatea cazurilor este necesar să se cunoască natura erorii (de exemplu pentru a fi scrisă într-un fișier de logging, pentru a fi consultată mai târziu).

Acest lucru se face specificând tipul excepției care ar trebui tratate în blocul *catch*:

```

using System;
public class Test
{
    public static void Main( )
    {
        Test t = new Test( );
        t.TestFunc( );
    }
    //incearca sa imparta doua numere
    public void TestFunc( )
    {
        try
        {
            double a = 5;
            double b = 0;
            Console.WriteLine (‘‘{0} / {1} = {2}’’, a, b, DoDivide(a,b));
        }
        //cel mai derivat tip de exceptie se specifica primul
        catch (System.DivideByZeroException)
        {
            Console.WriteLine(‘‘DivideByZeroException caught!’’);
        }
        catch (System.ArithmeticException)
        {
            Console.WriteLine(‘‘ArithmeticException caught!’’);
        }
        //Tipul mai general de exceptie este ultimul
        catch
        {
            Console.WriteLine(‘‘Unknown exception caught!’’);
        }
    }
    //efectueaza impartirea daca se poate
    public double DoDivide(double a, double b)
    {
        if (b == 0)
            throw new System.DivideByZeroException( );
        if (a == 0)
            throw new System.ArithmeticException( );
        return a/b;
    }
}

```

```
}
```

În exemplul de mai sus s-a convenit ca o împărțire cu numitor 0 să ducă la o excepție *System.DivideByZeroException*, iar o împărțire cu numărător 0 să ducă la apariția unei excepții de tip *System.ArithmeticException*. Este posibilă specificarea mai multor blocuri de tratare a excepțiilor. Aceste blocuri sunt parcurse în ordinea în care sunt specificate, iar primul tip care se potrivește cu excepția aruncată (în sensul că tipul excepție specificat este fie exact tipul obiectului aruncat, fie un tip de bază al acestuia - din cauză de upcasting) este cel care va face tratarea excepției apărute. Ca atare, este important ca ordinea excepțiilor tratate să fie de la cel mai derivat la cel mai general. În exemplul anterior, *System.DivideByZeroException* este derivat din clasa *System.ArithmeticException*.

Blocul *finally*

Uneori, aruncarea unei excepții și golirea stivei până la blocul de tratare a excepției poate să nu fie o idee bună. De exemplu, dacă excepția apare atunci când un fișier este deschis (și închiderea lui se poate face doar în metoda curentă), atunci ar fi util ca să se închidă fișierul înainte ca să fie preluat controlul de către metoda apelantă. Altfel spus, ar trebui să existe o garanție că un anumit cod se va executa, indiferent dacă totul merge normal sau apare o excepție. Acest lucru se face prin intermediul blocului *finally*, care se va executa în orice situație. Existența acestui bloc elimină necesitatea existenței blocurilor *catch* (cu toate că și acestea pot să apară).

Exemplu:

```
using System;
public class Test
{
    public static void Main( )
    {
        Test t = new Test( );
        t.TestFunc( );
    }
    public void TestFunc( )
    {
        try
        {
            Console.WriteLine(“Open file here”);
            double a = 5;
            double b = 0;
```

```

        Console.WriteLine (“{0} / {1} = {2}”, a, b, DoDivide(a,b));
        Console.WriteLine (“This line may or may not print”);
    }
    finally
    {
        Console.WriteLine (“Close file here.”);
    }
}
public double DoDivide(double a, double b)
{
    if (b == 0)
        throw new System.DivideByZeroException( );
    if (a == 0)
        throw new System.ArithmeticException( );
    return a/b;
}
}

```

În exemplul de mai sus, mesajul “Close file here” se va afișa indiferent de ce parametri se transmit metodei *DoDivide()*.

La aruncarea unei excepții se poate particulariza obiectul care se aruncă:

```

if (b == 0)
{
    DivideByZeroException e = new DivideByZeroException( );
    e.HelpLink = “http://www.greselifatale.com”;
    throw e;
}

```

iar când excepția este prinsă, se poate prelucra informația:

```

catch (System.DivideByZeroException e)
{
    Console.WriteLine( “DivideByZeroException!
        goto {0} and read more”, e.HelpLink);
}

```

Crearea propriilor excepții

În cazul în care suita de excepții predefinite nu este suficientă, programatorul își poate construi propriile tipuri. Se recomandă ca acestea să fie

derivate din *System.ApplicationException*, care este derivată direct din *System.Exception*. Se indică această derivare deoarece astfel se face distincție între excepțiile aplicație și cele sistem (cele aruncate de către CLR).

Exemplu:

```
using System;
public class MyCustomException : System.ApplicationException
{
    public MyCustomException(string message): base(message)
    {
    }
}

public class Test
{
    public static void Main( )
    {
        Test t = new Test( );
        t.TestFunc( );
    }

    public void TestFunc( )
    {
        try
        {
            double a = 0;
            double b = 5;
            Console.WriteLine (‘‘{0} / {1} = {2}’’, a, b, DoDivide(a,b));
            Console.WriteLine (‘‘This line may or may not print’’);
        }
        catch (System.DivideByZeroException e)
        {
            Console.WriteLine(‘‘DivideByZeroException! Msg: {0}’’,
                e.Message);
            Console.WriteLine(‘‘HelpLink: {0}’’,
                e.HelpLink);
        }
        catch (MyCustomException e)
        {
            Console.WriteLine(‘‘\nMyCustomException! Msg: {0}’’,
                e.Message);
        }
    }
}
```

```

        Console.WriteLine('\nHelpLink: {0}\n',
            e.HelpLink);
    }
    catch
    {
        Console.WriteLine('Unknown exception caught');
    }
}

public double DoDivide(double a, double b)
{
    if (b == 0)
    {
        DivideByZeroException e = new DivideByZeroException( );
        e.HelpLink= 'http://www.greselifatale.com';
        throw e;
    }
    if (a == 0)
    {
        MyCustomException e = new MyCustomException( 'Can't have
            zero divisor');
        e.HelpLink = 'http://www.greselifatale.com/NoZeroDivisor.htm';
        throw e;
    }
    return a/b;
}
}

```

Rearuncarea excepțiilor

Este perfect posibil ca într-un bloc de tratare a excepțiilor să se facă o tratare primară a excepției, după care să se arunce mai departe o altă excepție, de același tip sau de tip diferit (sau chiar excepția originală). Dacă se dorește ca această excepție să păstreze cumva în interiorul ei excepția originală, atunci constructorul permite înglobarea unei referințe la aceasta; această referință va fi accesibilă prin intermediul proprietății *InnerException*:

```

using System;
public class MyCustomException : System.ApplicationException
{
    public MyCustomException(string message,Exception inner):

```

```
        base(message,inner)
    {
    }
}
public class Test
{
    public static void Main( )
    {
        Test t = new Test( );
        t.TestFunc( );
    }
    public void TestFunc( )
    {
        try
        {
            DangerousFunc1( );
        }
        catch (MyCustomException e)
        {
            Console.WriteLine('\n{0}', e.Message);
            Console.WriteLine('Retrieving exception history...');
            Exception inner = e.InnerException;
            while (inner != null)
            {
                Console.WriteLine('{0}',inner.Message);
                inner = inner.InnerException;
            }
        }
    }
    public void DangerousFunc1( )
    {
        try
        {
            DangerousFunc2( );
        }
        catch(System.Exception e)
        {
            MyCustomException ex = new MyCustomException('E3 -
                Custom Exception Situation!',e);
            throw ex;
        }
    }
}
```

```

    }
    public void DangerousFunc2( )
    {
        try
        {
            DangerousFunc3( );
        }
        catch (System.DivideByZeroException e)
        {
            Exception ex =
            new Exception("E2 - Func2 caught divide by zero",e);
            throw ex;
        }
    }
    public void DangerousFunc3( )
    {
        try
        {
            DangerousFunc4( );
        }
        catch (System.ArithmeticException)
        {
            throw;
        }
        catch (System.Exception)
        {
            Console.WriteLine("Exception handled here.");
        }
    }
    public void DangerousFunc4( )
    {
        throw new DivideByZeroException("E1 - DivideByZero Exception");
    }
}

```

6.3.3 Reîncercarea codului

Se poate pune întrebarea: cum se procedează dacă se dorește revenirea la codul care a produs excepția, după tratarea ei? Există destule situații în care reexecutarea acestui cod este dorită: să ne gândim de exemplu la cazul în care într-o fereastră de dialog se specifică numele unei fișier ce trebuie

procesat, numele este introdus greșit și se dorește ca să se permită corectarea numelui. Un alt exemplu clasic este cazul în care autorul unei metode știe că o operație poate să eșueze periodic – de exemplu din cauza unui timeout pe rețea – dar vrea să reîncerce operația de n ori înainte de a semnala eroare.

În această situație se poate defini o etichetă înaintea blocului **try** la care să se permită saltul printr-un **goto**. Următorul exemplu permite unui utilizator să specifice de maxim trei ori numele unui fișier ce se procesează, cu revenire în cazul erorii.

```
using System;
using System.IO;

class Retry
{
    static void Main()
    {
        StreamReader sr;

        int attempts = 0;
        int maxAttempts = 3;

        GetFile:
        Console.WriteLine("\n[Attempt #{0}] Specify file " +
            "to open/read: ", attempts+1);
        string fileName = Console.ReadLine();

        try
        {
            sr = new StreamReader(fileName);

            Console.WriteLine();

            string s;
            while (null != (s = sr.ReadLine()))
            {
                Console.WriteLine(s);
            }
            sr.Close();
        }
        catch(FileNotFoundException e)
```

```

        {
            Console.WriteLine(e.Message);
            if (++attempts < maxAttempts)
            {
                Console.Write("Do you want to select " +
                    "another file: ");
                string response = Console.ReadLine();
                response = response.ToUpper();
                if (response == "Y") goto GetFile;
            }
            else
            {
                Console.Write("You have exceeded the maximum " +
                    "retry limit ({0})", maxAttempts);
            }
        }

    }
    catch(Exception e)
    {
        Console.WriteLine(e.Message);
    }

    Console.ReadLine();
}
}

```

6.3.4 Compararea tehnicilor de manipulare a erorilor

Metoda standard de tratare a erorilor a fost în general returnarea unui cod de eroare către metoda apelantă. Ca atare, apelantul are sarcina de a descifra acest cod de eroare și să reacționeze în consecință. Însă așa cum se arată mai jos, tratarea excepțiilor este superioară acestei tehnici din mai multe motive.

Neconsiderarea codului de retur

Apelul unei funcții care returnează un cod de eroare poate fi făcut și fără a utiliza efectiv codul returnat, scriind doar numele funcției cu parametrii de apel. Dacă de exemplu pentru o anumită procesare se apelează metoda A (de exemplu o deschidere de fișier) după care metoda B (citirea din fișier), se

poate ca în *A* să apară o eroare care nu este luată în considerare; apelul lui *B* este deja sortit eșecului, pentru că buna sa funcționare depinde de efectele lui *A*. Dacă însă metoda *A* aruncă o excepție, atunci nici măcar nu se mai ajunge la apel de *B*, deoarece CLR-ul va pasa execuția unui bloc `catch/finally`. Altfel spus, nu se permite o propagare a erorilor.

Manipularea erorii în contextul adecvat

În cazul în care o metodă *A* apelează alte metode B_1, \dots, B_n , este posibil ca oricare din aceste *n* metode să cauzeze o eroare (și să returneze cod adecvat); tratarea erorii din exteriorul lui *A* este dificilă în acest caz, deoarece ar trebui să se cerceteze toate codurile de eroare posibile pentru a determina motivul apariției erorii. Dacă se mai adaugă și apelul de metodă B_{n+1} în interiorul lui *A*, atunci orice apel al lui *A* trebuie să includă suplimentar și verificarea pentru posibilitatea ca B_{n+1} să fi cauzat o eroare. Ca atare, costul menținerii codului crește permanent, ceea ce are un impact negativ asupra TCO-ului⁶

Folosind tratarea excepțiilor, aruncând excepții cu mesaje de eroare explicite sau excepții de un anumit tip (definit de programator) se poate trata mult mai convenabil o situație deosebită. Mai mult decât atât, introducerea apelului lui B_{n+1} în interiorul lui *A* nu reclamă modificare suplimentară, deoarece tipul de excepție aruncat de B_{n+1} este deja tratat (desigur, se presupune că se definește un tip excepție sau o ierarhie de excepții creată convenabil).

Ușurința citirii codului

Pentru comparație, se poate scrie un cod care realizează procesarea conținutului unui fișier folosind coduri de eroare returnate sau excepții. În primul caz, soluția va conține cod de prelucrare a conținutului mixat cu cod de verificare și reacție pentru diferitele cazuri de excepție. Codul în al doilea caz este mult mai scurt, mai ușor de înțeles, de menținut, de corectat și extins.

Aruncarea de excepții din constructori

Nimic nu oprește ca o situație deosebită să apară într-un apel de constructor. Tehnica verificării codului de retur nu mai funcționează aici, deoarece un constructor nu returnează valori. Folosirea excepțiilor este în acest caz aproape de înlocuit.

⁶Total Cost of Ownership.

6.3.5 Sugestie pentru lucrul cu excepțiile

În Java, programatorii trebuie să declare că o metodă poate arunca o excepție și să o declare explicit într-o listă astfel încât un apelant să știe că se poate aștepta la primirea ei. Această cunoaștere în avans permite conceperea unui plan de lucru cu fiecare dintre ele, preferabil decât să se prindă oricare dintre ele cu un *catch* generic. În cazul .NET se sugerează să se mențină o documentație cu excepțiile care pot fi aruncate de fiecare metodă.

Curs 7

ADO.NET

7.1 Ce reprezintă ADO.NET?

ADO.NET reprezintă o parte componentă a lui .NET Framework ce permite aducerea, manipularea și modificarea datelor. În mod normal, o sursă de date poate să fie o bază de date, dar de asemenea un fișier text sau Excel sau XML sau Access. Lucrul se poate face fie conectat, fie deconectat de la sursa de date. Deși există variate modalități de lucru cu bazele de date, ADO.NET se impune tocmai prin faptul că permite lucrul deconectat de la baza de date, integrarea cu XML, reprezentarea comună a datelor cu posibilitatea de a combina date din variate surse, toate pe baza unor clase .NET.

Faptul că se permite lucrul deconectat de la sursa de date rezolvă următoarele probleme:

- menținerea conexiunilor la baza de date este o operație costisitoare. O bună parte a lățimii de bandă este menținută ocupată pentru niște procesări care nu necesită neapărat conectare continuă
- probleme legate de scalabilitatea aplicației: se poate ca serverul de baze de date să lucreze ușor cu 5-10 conexiuni menținute, dar dacă numărul acestora crește aplicația poate să reacționeze extrem de lent
- pentru unele servere se impun clauze asupra numărului de conexiuni ce se pot folosi simultan.

Toate acestea fac ca ADO.NET să fie o tehnologie mai potrivită pentru dezvoltarea aplicațiilor Internet decât cele precedente (e.g. ADO, ODBC).

Pentru o prezentare a metodelor de lucru cu surse de date sub platformă Windows se poate consulta [6].

Vom exemplifica în cele ce urmează preponderent pe bază de date Microsoft SQL 2005 Express Edition, ce se poate descărca gratuit de pe site-ul Microsoft.

7.2 Furnizori de date în ADO.NET

Din cauza existenței mai multor tipuri de surse de date (de exemplu, a mai multor producători de servere de baze de date) e nevoie ca pentru fiecare tip major de protocol de comunicare să se folosească o bibliotecă specializată de clase. Toate aceste clase implementează niște interfețe bine stabilite, ca atare trecerea de la un SGBD la altul se face cu eforturi minore (dacă codul este scris ținând cont de principiile POO).

Există următorii furnizori de date¹ (lista nu este completă):

Tabelul 7.1: Furnizori de date.

Nume furnizor	Prefix API	Descriere
ODBC Data Provider	Odbc	Surse de date cu interfață ODBC (baze de date vechi)
OleDb Data Provider	OleDb	Surse de date care expun o interfață OleDb, de exemplu Access și Excel sau SQL Server versiune mai veche de 7.0
Oracle Data Provider	Oracle	SGBD Oracle
SQL Data Provider	Sql	Pentru interacțiune cu Microsoft SQL Server 7.0, 2000, 2005
Borland Data Provider	Bdp	Acces generic la SGBD-uri precum Interbase, SQL Server, IBM DB2, Oracle
MySql	MySql	SGBD MySql

Prefixele trecute în coloana "Prefix" sunt folosite pentru clasele de lucru specifice unui anumit furnizor ADO.NET: de exemplu, pentru o conexiune SQL Server se va folosi clasa SqlConnection.

7.3 Componentele unui furnizor de date

Fiecare furnizor de date ADO.NET constă în patru componente: *Connection*, *Command*, *DataReader*, *DataAdapter*. Arhitectura ADO.NET este

¹Engl: Data Providers

prezentată în figura 7.1

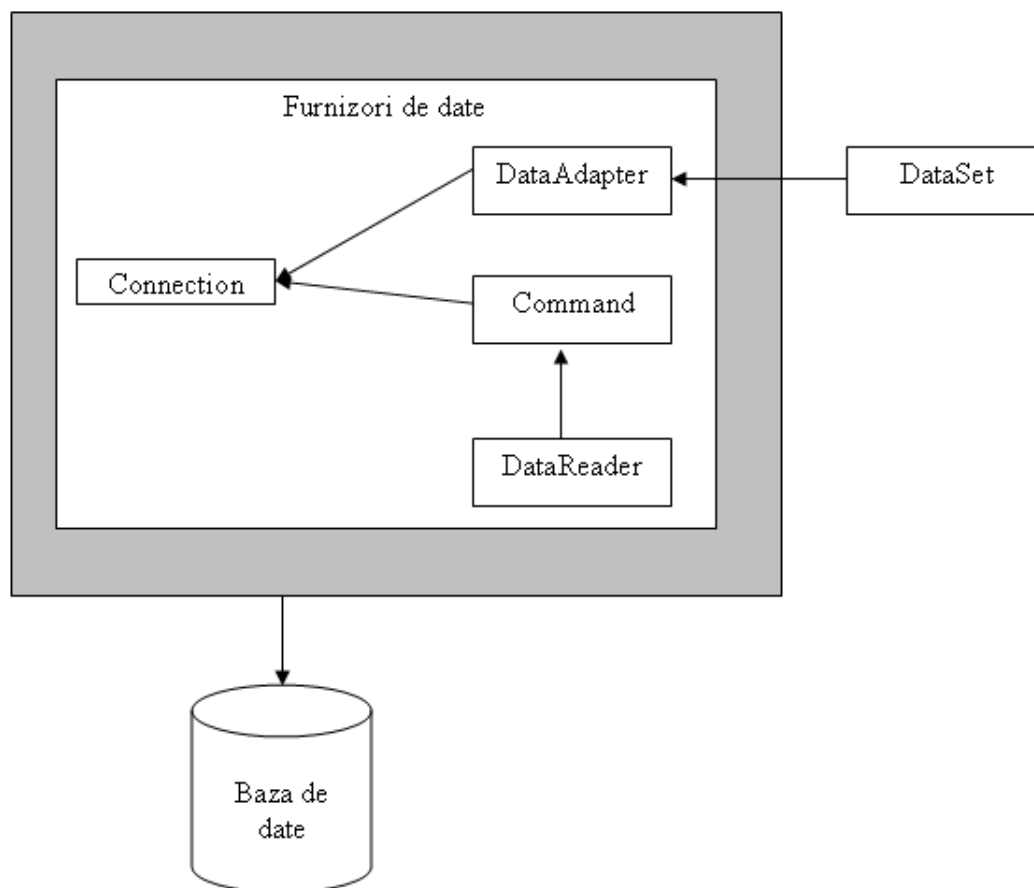


Figura 7.1: Principalele clase ADO.NET

Mai jos sunt date descrieri succinte ale claselor cel mai des utilizate.

7.3.1 Clasele *Connection*

Sunt folosite pentru a reprezenta o conexiune la surse de date specifice. Ele conțin date specifice conexiunii, cum ar fi locația sursei de date, numele și parola contului de acces, etc. În plus au metode pentru deschiderea și închiderea conexiunilor, pornirea unei tranzacții sau setarea perioadei de time-out. Stau la baza oricărei accesări de servicii de pe server.

7.3.2 Clasele *Command*

Sunt folosite pentru a executa diferite comenzi pe baza de date (SELECT, INSERT, UPDATE, DELETE) și pentru a furniza un obiect de tip *DataReader* sau *DataSet*. Pot fi folosite pentru apelarea de proceduri stocate aflate pe server. Ele permit scrierea de interogări SQL parametrizate sau specificarea parametrilor pentru procedurile stocate.

7.3.3 Clasele *DataReader*

Permit navigarea de tip forward-only, read-only în mod conectat la sursa de date. Se obțin pe baza unui obiect de tip *Command* prin apelul metodei *ExecuteReader()*. Accesul rezultat este extrem de rapid cu minim de resurse consumate.

7.3.4 Clasele *DataAdapter*

Ultima componentă principală a unui furnizor de date .NET este *DataAdapter*. Funcționează ca o punte între sursa de date și obiecte de tip *DataSet* deconectate, permițând efectuarea operațiilor pe baza de date. Conține referințe către obiecte de tip *Connection* și deschid / închid singure conexiunea la baza de date. În plus, un *DataAdapter* conține referințe către patru comenzi pentru selectare, ștergere, modificare și adăugare la baza de date.

7.3.5 Clasa *DataSet*

Această clasă nu este parte a unui furnizor de date .NET ci independentă de particularitățile de conectare și lucru cu o bază de date anume. Prezintă marele avantaj că poate să lucreze deconectat de la sursa de date, facilitând stocarea și modificarea datelor local, apoi reflectarea acestor modificări în baza de date. Un obiect *DataSet* este de fapt un container de tabele și relații între tabele. Folosește serviciile unui obiect de tip *DataAdapter* pentru a-și procura datele și a trimite modificările înapoi către baza de date. Datele sunt stocate de un *DataSet* în format XML; același format este folosit pentru transportul datelor.

7.4 Obiecte *Connection*

Clasele de tip *Connection* pun la dispoziție tot ceea ce e necesar pentru conectarea la baze de date. Este primul obiect cu care un programator ia contact atunci când încearcă să folosească un furnizor de date .NET. Înainte

ca o comandă să fie executată pe o bază de date trebuie stabilită datele de conectare și descisă conexiunea.

Orice clasă de tip conexiune (din orice furnizor de date) implementează interfața *IDbConnection*. De exemplu, clasele *SqlConnection* (folosită pentru conectare la server Microsoft SQL Server 2000) sau *OleDbConnection* (folosită pentru conectare la fișiere .mdb din Access sau Excel) implementează *IDbConnection*.

Pentru deschiderea unei conexiuni se poate proceda ca mai jos:

```
using System.Data.SqlClient; //spatiul de nume SqlConnection
...
SqlConnection cn = new SqlConnection(@"Data Source=
localhost\squlexpress;Database=Northwind;User ID=sa;
Password=parola");
cn.Open();
...
```

Mai sus s-a specificat numele calculatorului pe care se află instalat severul MSSQL de baze de date ("serverBD"), baza de date la care se face conectarea ("Northwind"), contul SQL cu care se face accesul ("sa") și parola pentru acest cont ("parola").

Pentru conectarea la un fișier Access *Northwind.mdb* aflat în directorul c:\lucru se folosește un obiect de tipul *OleDbConnection* sub forma:

```
using System.Data.OleDb; //spatiul de nume OleDb
...
OleDbConnection cn = new OleDbConnection(
@"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
C:\Lucru\Northwind.mdb");
cn.Open();
...
```

S-a specificat furnizorul de date (Microsoft.Jet.OLEDB.4.0 pentru fișier Access) precum și locul unde se află sursa de date (C:\Lucru\Northwind.mdb).

Vom enumera principalele proprietăți, metode și evenimente pentru un obiect de tip *Connection*.

7.4.1 Proprietăți

1. *ConnectionString*: de tip *String*, cu accesorii de *get* și *set*; această proprietate definește un string valid care permite identificarea tipului și locației sursei de date la care se face conectarea și eventual contul și

parola de acces. Acest string conține lista de parametri necesari pentru conectare sub forma numeParametru=valoare, separați prin punct și virgulă. Parametrii sunt:

- *provider*: se specifică furnizorul de date pentru conectarea la sursa de date. Acest furnizor trebuie precizat doar dacă se folosește OLE DB .NET Data Provider, însă nu se specifică pentru conectare la SQL Server.
 - *Data Source*: se specifică numele serverului de baze de date sau numele fișierului de date.
 - *Initial Catalog* (sinonim cu *Database*): specifică numele baze de date. Baza de date trebuie să se găsească pe serverul dat în *Data Source*.
 - *User ID* (sinonim cu *uid*): specifică un nume de utilizator care are acces de loginare la server.
 - *Password* (sinonim cu *pwd*): specifică parola contului de mai sus.
2. *ConnectionTimeout*: de tip `int`, cu accesoriu de *get*, valoare implicită 15; specifică numărul de secunde pentru care un obiect de conexiune ar trebui să aștepte pentru realizarea conectării la server înainte de a se genera o excepție. Se poate specifica o valoare diferită de 15 în *ConnectionString* folosind parametrul *Connect Timeout*:
- ```
SqlConnection cn = new SqlConnection("Data Source=serverBD;
Database=Northwind;User ID=sa;Password=parola;
Connect Timeout=30");
```
- Se poate specifica pentru *Connect Timeout* valoarea 0 cu semnificația "așteaptă oricât", dar se sugerează să nu se procedeze în acest mod.
3. *Database*: atribut de tip `string`, read-only, returnează numele bazei de date la care s-a făcut conectarea. Folosită pentru a arăta unui utilizator care este baza de date pe care se face operarea.
4. *Provider*: atribut de tip `string`, read-only, returnează furnizorul OLE DB.
5. *ServerVersion*: atribut de tip `string`, read-only, returnează versiunea de server la care s-a făcut conectarea.
6. *State*: atribut de tip enumerare `ConnectionState`, read-only, returnează starea curentă a conexiunii. Valorile posibile sunt: *Broken*, *Closed*, *Connecting*, *Executing*, *Fetching*, *Open*.

### 7.4.2 Metode

1. *Open()*: deschide o conexiune la baza de date
2. *Close()*, *Dispose()*: închid conexiunea și eliberează toate resursele alocate pentru ea
3. *BeginTransaction()*: pentru executarea unei tranzacții pe baza de date; la sfârșit se apelează *Commit()* sau *Rollback()*.
4. *ChangeDatabase()*: se modifică baza de date la care se vor face conexiunile. Noua bază de date trebuie să existe pe același server ca precedentă.
5. *CreateCommand()*: creează un obiect de tip *Command* valid (care implementează interfața *IDbCommand*) asociat cu conexiunea curentă.

### 7.4.3 Evenimente

Un obiect de tip conexiune poate semnala două evenimente:

- evenimentul *StateChange*: apare atunci când se schimbă starea conexiunii. Event-handlerul este de tipul delegat *StateChangeEventHandler*, care spune care sunt stările între care s-a făcut tranziția.
- evenimentul *InfoMessage*: apare atunci când furnizorul trimite un avertisment sau un mesaj informațional către client.

### 7.4.4 Stocarea stringului de conexiune în fișier de configurare

În general este contraindicat ca stringul de conexiune să fie scris direct în cod; modificarea datelor de conectare (de exemplu parola pe cont sau sursa de date) ar însemna recompilarea codului.

.NET Framework permite menținerea unor perechi de tipul chei-valoare, specifice aplicației; fișierul este de tip XML. Pentru aplicațiile Web fișierul se numește *web.config*, pentru aplicațiile de tip consolă fișierul de configurare are extensia *config* și numele aplicației, iar pentru aplicațiile Windows acest fișier nu există implicit, dar se adaugă: Project->Add new item->Application Configuration File, implicit acesta având numele *App.config*. În fie care caz, structura fișierului XML de configurare este similară. Implicit, acesta arată astfel:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

În interiorul elementului rădăcină *configuration* se va introduce elementul *appSettings*, care va conține oricâte perechi cheie-valoare în interiorul unui atribut numit *add*, precum mai jos:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <appSettings>
 <add key="constring"
 value="Data Source=localhost\\sqlexpress;database=Northwind;
 User ID=sa;pwd=parola"/>
 </appSettings>
</configuration>
```

Clasele necesare pentru accesarea fișierului de configurare se găsesc în spațiul de nume *System.Configuration*. Pentru a se putea folosi acest spațiu de nume trebuie să se adauge o referință la assembly-ul care conține deaceastă clasă: din Solutin explorer click dreapta pe proiect->Add reference...->se alege tab-ul .NET și de acolo System.Configuration. Utilizarea stringului de conexiune definit anterior se face astfel:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
public class UsingConfigSettings
{
 public static void Main()
 {
 SqlConnection con = new SqlConnection(
 ConfigurationManager.AppSettings["constring"];
 //se lucreaza cu conexiunea
 con.Close();
 }
}
```

Este posibil ca într-o aplicație să se folosească mai multe conexiuni, motiv pentru care se sugerează ca în loc de varianta precedentă să se folosească elementul XML `<connectionStrings>`:



```
<configuration>
 <appSettings>...</appSettings>
 <connectionStrings>
 <add name="SqlProviderPubs" connectionString =
 "Data Source=localhost\sqlexpress;uid=sa;pwd=;
 Initial Catalog=Pubs"/>
 <add name="OleDbProviderPubs" connectionString =
 "Provider=SQLOLEDB.1;Data Source=localhost;uid=sa;pwd=;
 Initial Catalog=Pubs"/>
 </connectionStrings>
</configuration>
```

Preluarea unui string de conexiune se face prin:

```
string cnStr =
 ConfigurationManager.ConnectionStrings["SqlProviderPubs"].ConnectionString;
```

### 7.4.5 Gruparea conexiunilor

Gruparea conexiunilor<sup>2</sup> reprezintă reutilizarea resurselor de tip conexiune la o bază de date. Atunci când se creează o grupare de conexiuni se generează automat mai multe obiecte de tip conexiune, acest număr fiind egal cu minimul setat pentru gruparea respectivă. O nouă conexiune este creată dacă toate conexiunile sunt ocupate și se cere o nouă conexiune. Dacă dimensiunea maxim setată a grupării este atinsă, atunci nu se va mai crea o conexiune nouă, ci se va pune într-o coadă de așteptare. Dacă așteptarea durează mai mult decât este precizat în valoarea de Timeout se va arunca o excepție. Pentru a returna o conexiune la grupare trebuie apelată metoda *Close()* sau *Dispose()* pentru acea conexiune.

Sursele de date .NET administrează automat acest aspect, degrevându-l pe programator de acest aspect ce nu ține de logica aplicației. La dorință comportamentul implicit se poate modifica prin intermediul conținutului stringului de conectare.

### 7.4.6 Mod de lucru cu conexiunile

Se cere ca o conexiune să fie întotdeauna închisă (și dacă se poate, cât mai repede posibil). Ca atare, este de preferat ca să se aplice o schemă de lucru de tipul:

---

<sup>2</sup>Engl: connection pooling

```
IDBConnection con = ...
try
{
 //deschidere conexiune
 //lucru pe baza
}
catch(Exception e)
{
 //tratare de exceptie
}
finally
{
 con.Close();
}
```

Deoarece se garantează ca blocul `finally` este executat indiferent dacă apare sau nu o excepție, în cazul de mai sus se va închide în mod garantat conexiunea.

## 7.5 Obiecte *Command*

Un clasă de tip *Command* dată de un furnizor .NET trebuie să implementeze interfața *IDbCommand*, ca atare toate vor asigura un set de servicii bine specificat. Un asemenea obiect este folosit pentru a executa comenzi pe baza de date: SELECT, INSERT, DELETE, UPDATE sau apel de proceduri stocate (dacă SGBD-ul respectiv știe acest lucru). Comanda se poate executa numai dacă s-a deschis o conexiune la baza de date.

Exemplu:

```
SqlConnection con = new SqlConnection(
 ConfigurationManager.AppSettings["constring"];
SqlCommand cmd = new SqlCommand("SELECT * FROM Customers", con);
```

Codul care utilizează o comandă pentru lucrul cu fișiere mdb sau xls ar fi foarte asemănător, cu diferența că în loc de *SqlCommand* se folosește *OleDbCommand*, din spațiul de nume *System.Data.OleDb*. Nu trebuie modificat altceva, deoarece locația sursei de date se specifică doar la conexiune.

Se observă că obiectul de conexiune este furnizat comenzii create. Enumerăm mai jos principalele proprietăți și metode ale unui obiect de tip comandă.

### 7.5.1 Proprietăți

1. *CommandText*: de tip *String*, cu ambii accesorii; conține comanda SQL sau numele procedurii stocate care se execută pe sursa de date.
2. *CommandTimeout*: de tip *int*, cu ambii accesorii; reprezintă numărul de secunde care trebuie să fie așteptat pentru executarea interogării. Dacă se depășește acest timp, atunci se aruncă o excepție.
3. *CommandType*: de tip *CommandType* (enumerare), cu ambii accesorii; reprezintă tipul de comandă care se execută pe sursa de date. Valorile pot fi:
  - *CommandType.StoredProcedure* - interpretează comanda conținută în proprietatea *CommandText* ca o un apel de procedură stocată definită în baza de date.
  - *CommandType.Text* - interpretează comanda ca fiind o comandă SQL clasică; este valoarea implicită.
  - *CommandType.TableDirect* - momentan disponibil numai pentru furnizorul OLE DB (deci pentru obiect de tip *OleDbCommand*); dacă proprietatea *CommandType* are această valoare, atunci proprietatea *CommandText* este interpretată ca numele unui tabel pentru care se aduc toate liniile și coloanele la momentul executării.
4. *Connection* - proprietate de tip *System.Data.[.NET Data Provider].PrefixConnection*, cu ambii accesorii; conține obiectul de tip conexiune folosit pentru legarea la sursa de date; "Prefix" este prefixul asociat furnizorului respectiv (a se vedea tabelul 7.1).
5. *Parameters* - proprietate de tip *System.Data.[.NET Data Provider].PrefixParameterCollection*, read-only; returnează o colecție de parametri care s-au transmis comenzii; această listă a fost creată prin apelul metodei *CreateParameter()*. "Prefix" reprezintă același lucru ca mai sus.
6. *Transaction* - proprietate de tip *System.Data.[.NET Data Provider].PrefixTransaction*, read-write; permite accesul la obiectul de tip tranzacție care se cere a fi executat pe sursa de date.

### 7.5.2 Metode

1. *Constructori* - un obiect de tip comandă poate fi creat și prin intermediul apelului de constructor; de exemplu un obiect *SqlCommand* se

poate obține astfel:

```
SqlCommand cmd;
cmd = new SqlCommand();
cmd = new SqlCommand(string.CommandText);
cmd = new SqlCommand(string.CommandText, SqlConnection con);
cmd = new SqlCommand(string.CommandText, SqlConnection con,
 SqlTransaction trans);
```

2. *Cancel()* - încearcă să oprească o comandă dacă ea se află în execuție. Dacă nu se află în execuție, sau această încercare nu are nici un efect nu se întâmplă nimic.
3. *Dispose()* - distruge obiectul comandă.
4. *ExecuteNonQuery()* - execută o comandă care nu returnează un set de date din baza de date; dacă comanda a fost de tip INSERT, UPDATE, DELETE, se returnează numărul de înregistrări afectate. Dacă nu este definită conexiunea la baza de date sau aceasta nu este deschisă, se aruncă o excepție de tip *InvalidOperationException*.

Exemplu:

```
SqlConnection con = new SqlConnection(
 ConfigurationManager.AppSettings["constring"]);
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "DELETE FROM Customers WHERE CustomerID = 'SEVEN'";
cmd.Connection = con;
con.Open();
Console.WriteLine(cmd.ExecuteNonQuery().ToString());
con.Close()
```

În exemplul de mai sus se returnează numărul de înregistrări care au fost șterse.

5. *ExecuteReader()* - execută comanda conținută în proprietatea *CommandText* și se returnează un obiect de tip *IDataReader* (e.g. *SqlDataReader* sau *OleDbDataReader*).

Exemplu: se obține conținutul tabelii Customers într-un obiect de tip *SqlDataReader* (se presupune că baza de date se stochează pe un server MSSQL):

```
SqlConnection con = new SqlConnection(
 ConfigurationManager.AppSettings["constring"]);
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "SELECT * FROM Customers";
cmd.Connection = con;
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
 Console.WriteLine("{0} - {1}",
 reader.GetString(0),
 reader.GetString(1));
}
reader.Close();
con.Close();
```

Metoda *ExecuteReader()* mai poate lua un argument opțional de tip enumerare *CommandBehavior* care descrie rezultatele și efectul asupra bazei de date:

- *CommandBehavior.CloseConnection* - conexiunea este închisă atunci când obiectul de tip *IDataReader* este închis.
- *CommandBehavior.KeyInfo* - comanda returnează informație despre coloane și cheia primară.
- *CommandBehavior.SchemaOnly* - comanda returnează doar informație despre coloane.
- *CommandBehavior.SequentialAccess* - dă posibilitatea unui *DataReader* să manipuleze înregistrări care conțin câmpuri cu valori binare de mare întindere. Acest mod permite încărcarea sub forma unui flux de date folosind *GetChars()* sau *GetBytes()*.
- *CommandBehavior.SingleResult* - se returnează un singur set de rezultate
- *CommandBehavior.SingleRow* - se returnează o singură linie. De exemplu, dacă în codul anterior înainte de *while* obținerea obiectului *reader* s-ar face cu:

```
SqlDataReader reader = cmd.ExecuteReader(
 CommandBehavior.SingleRow);
```

atunci s-ar returna doar prima înregistrare din setul de date.

6. *ExecuteScalar()* - execută comanda conținută în proprietatea *CommandText*; se returnează valoarea primei coloane de pe primul rând a setului de date rezultat; folosit pentru obținerea unor rezultate de tip agregat ("SELECT COUNT(\*) FROM CUSTOMERS", de exemplu).
7. *ExecuteXmlReader()* - returnează un obiect de tipul *XmlReader* obținut din rezultatul interogării pe sursa de date.

Exemplu:

```
SqlCommand custCMD=new SqlCommand("SELECT * FROM Customers
 FOR XML AUTO, ELEMENTS", con);
System.Xml.XmlReader myXR = custCMD.ExecuteXmlReader();
```

### 7.5.3 Utilizarea unei comenzi cu o procedură stocată

Pentru a se executa pe server o procedură stocată definită în baza respectivă, este necesar ca obiectul comandă să aibe proprietatea *CommandType* la valoarea *CommandType.StoredProcedure* iar proprietatea *CommandText* să conțină numele procedurii stocate:

```
SqlConnection con = new SqlConnection(
 ConfigurationManager.AppSettings["constring"]);
SqlCommand cmd = new SqlCommand("Ten Most Expensive Products", con);
cmd.CommandType = CommandType.StoredProcedure;
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
 Console.WriteLine("{0} - {1}",
 reader.GetString(0), reader.GetDecimal(1));
}
reader.Close();
con.Close();
```

*Observație:* se remarcă în toate exemplele de mai sus faptul că fiecare conexiune se închide manual, printr-un apel de tipul *con.Close()*. Dacă conexiunea a fost folosită pentru un obiect de tip *DataRead* atunci acesta din urmă trebuie să fie și el închis, înaintea închiderii conexiunii. Dacă nu se face acest apel atunci conexiunea va fi ținută ocupată și nu va putea fi reutilizată.

### 7.5.4 Folosirea comenzilor parametrizate

Există posibilitatea de a rula cod SQL (interogări, proceduri stocate) pe server care să fie parametrizate. Orice furnizor de date .NET permite crearea obiectelor parametru care pot fi adăugate la o colecție de parametri ai comenzii. Valoarea acestor parametri se specifică fie prin numele lor (cazul *SqlParameter*), fie prin poziția lor (cazul *OleDbParameter*).

Exemplu: vom aduce din tabela *Customers* toate înregistrările care au în câmpul *Country* valoarea "USA".

```
SqlConnection con = new SqlConnection(
 ConfigurationManager.AppSettings["constring"]);
SqlCommand cmd = new
 SqlCommand("SELECT * FROM Customers WHERE Country=@country", con);
SqlParameter param = new SqlParameter("@country", SqlDbType.VarChar);
param.Value = "USA";
cmd.Parameters.Add(param);
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
 Console.WriteLine("{0} - {1}",
 reader.GetString(0), reader.GetString(1));
}
reader.Close();
con.Close();
```

Pentru parametrul creat s-a setat tipul lui (ca fiind tip SQL șir de caractere) și valoarea. De reținut faptul că numele parametrului se prefixează cu caracterul "@".

În cazul în care un parametru este de ieșire, acest lucru trebuie spus explicit folosind proprietatea *Direction* a parametrului respectiv:

```
SqlCommand cmd = new SqlCommand(
 "SELECT * FROM Customers WHERE Country = @country; " +
 "SELECT @count = COUNT(*) FROM Customers WHERE Country = @country",
 con);
SqlParameter param = new SqlParameter("@country", SqlDbType.VarChar);
param.Value = "USA";
cmd.Parameters.Add(param);
cmd.Parameters.Add(new SqlParameter("@count", SqlDbType.Int));
cmd.Parameters["@count"].Direction = ParameterDirection.Output;
```

```
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
 Console.WriteLine("{0} - {1}",
 reader.GetString(0),
 reader.GetString(1));
}
reader.Close();
Console.WriteLine("{0} - {1}", "Count",
 cmd.Parameters["@count"].Value.ToString());
con.Close();
```

Remarcăm următoarele:

- este posibil ca într-o comandă să se execute mai multe interogări
- pentru parametrul de ieșire numit “@count” trebuie făcută declarare de direcție; implicit un parametru este de intrare
- parametrii de ieșire sunt accesibili doar după închiderea obiectului de tip *DataReader*

## 7.6 Obiecte *DataReader*

Un obiect de tip *DataReader* este folosit pentru a citi date dintr-o sursă de date. Caracteristicile unei asemenea clase sunt:

1. implementează întotdeauna interfața *IDataReader*
2. se lucrează conectat la sursa de date - pe toată perioada cât este accesat un *DataReader* necesită conexiune activă
3. este de tip read-only; dacă se dorește modificarea datelor se poate folosi un *DataSet* + *DataAdapter*
4. este de tip forward-only - metoda de modificare a poziției curente este doar în direcția înainte; orice reîntoarcere presupune reluarea înregistrărilor (dacă programatorul nu implementează el singur un mecanism de coadă)

Avantajele utilizării acestui tip de obiecte sunt: accesul conectat, performanțele bune, consumul mic de resurse și tipizarea puternică.



### 7.6.1 Proprietăți

1. *IsClosed* - proprietate read-only, returnează *true* dacă obiectul este deschis, *false* altfel
2. *HasRows* - proprietate booleană read-only care spune dacă readerul conține cel puțin o înregistrare
3. *Item* - indexator care dă acces la câmpurile unei înregistrări
4. *FieldCount* - dă numărul de câmpuri din înregistrarea curentă

### 7.6.2 Metode

1. *Close()* - închide obiectul de citire și eliberează resursele client. Este obligatoriu apelul acestei metode înainte de închiderea conexiunii.
2. *GetBoolean()*, *GetByte()*, *GetChar()*, *GetDateTime()*, *GetDecimal()*, *GetDouble()*, *GetFloat()*, *GetInt16()*, *GetInt32()*, *GetInt64()*, *GetValue()*, *GetString()* returnează valorile câmpurilor din înregistrarea curentă. Preiau ca parametru indicele coloanei a cărei valoare se cere. *GetValue()* returnează un obiect de tip *Object*, pentru celelalte tipul returnat este descris de numele metodelor.
3. *GetBytes()*, *GetChars()* - returnează numărul de octeți / caractere citați dintr-un câmp ce stochează o structură de dimensiuni mari; primește ca parametri indicele de coloană (int), poziția din acea coloană de unde se va începe citirea, vectorul în care se face citirea, poziția în buffer de la care se depun datele citite, numărul de octeți/caractere ce urmează a fi citați.
4. *GetDataTypeName()* - returnează tipul coloanei specificat prin indice
5. *GetName()* - returnează numele coloanei
6. *IsDBNull()* - returnează *true* dacă în câmpul specificat prin index este o valoare de *NULL* (din baza de date)
7. *NextResult()* - determină trecerea la următorul rezultat, dacă aceasta există; în acest caz returnează *true*, altfel *false* (este posibil ca într-un *DataReader* să vină mai multe rezultate, provenind din interogări diferite)

8. *Read()* - determină trecerea la următoarea înregistrare, dacă aceasta există; în acest caz ea returnează *true*. Metoda trebuie chemată cel puțin o dată, deoarece inițial poziția curentă este înaintea primei înregistrări.

### 7.6.3 Crearea și utilizarea unui *DataReader*

Nu este posibil să se creeze un obiect de tip *DataReader* cu ajutorul unui constructor, ci prin intermediul unui obiect de tip *Command*, folosind apelul *ExecuteReader()* (a se vedea secțiunea 7.3.2). Pentru comanda respectivă se specifică instrucțiunea care determină returnarea setului de date precum și obiectul de conexiune. Această conexiune trebuie să fie deschisă înaintea apelului *ExecuteReader()*. Trecerea la următoarea înregistrare se face folosind metoda *Read()*. Când se dorește încetarea lucrului se închide reader-ul și conexiunea. Omiterea închiderii obiectului de tip reader va duce la imposibilitatea reutilizării conexiunii inițiale. După ce se închide acest *DataReader* este necesară și închiderea explicită a conexiunii (acest lucru nu mai e mandatoriu doar dacă la apelul metodei *ExecuteReader* s-a specificat *CommandBehavior.CloseConnection*). Dacă se încearcă refolosirea conexiunii fără ca readerul să fi fost închis se va arunca o excepție *InvalidOperationException*.

Exemplu:

```
SqlConnection conn = new SqlConnection (
 ConfigurationManager.AppSettings["constring"]);
SqlCommand selectCommand = new SqlCommand("select * from ORDERS", conn);
conn.Open ();
OleDbDataReader reader = selectCommand.ExecuteReader ();
while (reader.Read ())
{
 object id = reader["OrderID"];
 object date = reader["OrderDate"];
 object freight = reader["Freight"];
 Console.WriteLine ("{0}\t{1}\t\t{2}", id, date, freight);
}
reader.Close ();
conn.Close ();
```

Este perfect posibil ca un obiect de tip *DataReader* să aducă datele prin apelul unei proceduri stocate (de fapt invocarea acestei proceduri este făcută de către obiectul de tip *Command*).

Următoarele observații trebuie luate în considerare atunci când se lucrează cu un obiect *DataReader*:

- Metoda *Read()* trebuie să fie întotdeauna apelată înaintea oricărui acces la date; poziția curentă la deschidere este înaintea primei înregistrări.
- Întotdeauna apălați metoda *Close()* pe un *DataReader* și pe conexiunea asociată cât mai repede posibil; în caz contrar conexiunea nu poate fi reutilizată
- Procesarea datelor citite trebuie să se facă după închiderea conexiunii; în felul acesta conexiunea se lasă liberă pentru a putea fi reutilizată.

#### 7.6.4 Utilizarea de seturi de date multiple

Este posibil ca într-un *DataReader* să se aducă mai multe seturi de date. Acest lucru ar micșora numărul de apeluri pentru deschiderea unei conexiuni la stratul de date. Obiectul care permite acest lucru este chiar cel de tip *Command*:

```
string select = "select * from Categories; select * from customers";
SqlCommand command = new SqlCommand (select, conn);
conn.Open ();
SqlDataReader reader = command.ExecuteReader ();
```

Trecerea de la un set de date la altul se face cu metoda *NextResult()* a obiectului de tip *Reader*:

```
do
{
 while (reader.Read ())
 {
 Console.WriteLine ("{0}\t\t{1}", reader[0], reader[1]);
 }
}while (reader.NextResult ());
```

#### 7.6.5 Accesarea datelor într-o manieră sigură din punct de vedere a tipului

Să considerăm următoarea secvență de cod:

```
while (reader.Read ())
{
 object id = reader["OrderID"];
 object date = reader["OrderDate"];
 object freight = reader["Freight"];
```

```
 Console.WriteLine ("{0}\t{1}\t\t{2}", id, date, freight);
}
```

După cum se observă, este posibil ca valorile câmpurilor dintr-o înregistrare să fie accesate prin intermediul numelui coloanei (sau a indicelui ei, pornind de la 0). Dezavantajul acestei metode este că tipul datelor returnate este pierdut (fiind returnate obiecte de tip *Object*), trebuind făcută un downcasting pentru a utiliza din plin facilitățile tipului respectiv. Pentru ca acest lucru să nu se întâmple se pot folosi metodele *GetX<sub>Y</sub>* care returnează un tip specific de date:

```
while (reader.Read ())
{
 int id = reader.GetInt32 (0);
 DateTime date = reader.GetDateTime (3);
 decimal freight = reader.GetDecimal (7);
 Console.WriteLine ("{0}\t{1}\t\t{2}", id, date, freight);
}
```

Avantajul secvenței anterioare este că dacă se încearcă aducerea valorii unui câmp pentru care tipul nu este dat corect se aruncă o excepție *InvalidCastException*; altfel spus, accesul la date se face sigur din punct de vedere al tipului datelor.

Pentru a evita folosirea unor “constante magice” ca indici de coloană (precum mai sus: 0, 3, 7), se poate folosi următoarea strategie: indicii se obțin folosind apel de metodă *GetOrdinal* la care se specifică numele coloanei dorite:

```
private int OrderID;
private int OrderDate;
private int Freight;
...
OrderID = reader.GetOrdinal("OrderID");
OrderDate = reader.GetOrdinal("OrderDate");
Freight = reader.GetOrdinal("Freight");
...
reader.GetDecimal (Freight);
...
```

## Curs 8

# ADO.NET (2)

### 8.1 Obiecte *DataAdapter*

La fel ca și *Connection*, *Command*, *DataReader*, obiectele de tip *DataAdapter* fac parte din furnizorul de date .NET specific fiecărui tip de sursă de date. Scopul ei este să permită umplerea unui obiect *DataSet* cu date și reflectarea schimbărilor efectuate asupra acestuia înapoi în baza de date (*DataSet* permite lucrul deconectat de la baza de date).

Orice clasă de tipul *DataAdapter* (de ex *SqlDataAdapter* și *OleDbDataAdapter*) este derivată din clasa *DbDataAdapter* (clasă abstractă). Pentru orice obiect de acest tip trebuie specificată minim comanda de tip *SELECT* care să populeze un obiect de tip *DataSet*; acest lucru este stabilit prin intermediul proprietății *SelectCommand* de tip *Command* (*SqlCommand*, *OleDbCommand*, ...). În cazul în care se dorește și modificarea informațiilor din sursa de date (inserare, modificare, ștergere) trebuie specificate obiecte de tip comandă via proprietățile: *InsertCommand*, *UpdateCommand*, *DeleteCommand*.

Exemplu: mai jos se preiau înregistrările din 2 tabele: *Authors* și *TitleAuthor* și se trec într-un obiect de tip *DataSet* pentru a fi procesate ulterior.

```
using System;
using System.Data;
using System.Data.SqlClient;

class DemoDataSource
{
 static void Main()
 {
 SqlConnection conn = new SqlConnection(
```

```

 ConfigurationManager.AppSettings["constring"]);

 SqlDataAdapter daAuthors = new SqlDataAdapter("SELECT au_id,
 au_fname, au_lname FROM authors", conn);
 daAuthors.Fill(ds, "Author");

 SqlDataAdapter daTitleAuthor = new SqlDataAdapter("SELECT
 au_id, title_id FROM titleauthor", conn);
 daTitleAuthor.Fill(ds, "TitleAuthor");
 }
}

```

Prezentăm mai jos cele mai importante componente ale unei clase de tip *DataAdapter*.

### 8.1.1 Metode

1. *Constructorii* de la cei impliciți până la cei în care se specifică o comandă de tip *SELECT* și conexiunea la sursa de date. Pentru un obiect de tip *SqlDataAdapter* se poate crea o instanță în următoarele moduri:

```
SqlDataAdapter da = new SqlDataAdapter();
```

sau:

```
SqlCommand cmd = new SqlCommand("SELECT * FROM Employees");
SqlDataAdapter da = new SqlDataAdapter(cmd);
```

sau:

```
String strCmd = "SELECT * FROM Employees";
String strConn = "...";
SqlDataAdapter da = new SqlDataAdapter(strCmd, strConn);
```

2. *Fill()* – metodă polimorfică, permițând umplerea unei tabele dintr-un obiect de tip *DataSet* cu date. Permite specificarea obiectului *DataSet* în care se depun datele, eventual a numelui tablei din acest *DataSet*, numărul de înregistrare cu care să se înceapă popularea (prima având indicele 0) și numărul de înregistrări care urmează a fi aduse. Returnează de fiecare dată numărul de înregistrări care au fost aduse din bază. În clipa în care se apelează *Fill()* se procedează astfel:

- (a) Se deschide conexiunea (dacă ea nu a fost explicit deschisă)
- (b) Se aduc datele și se populează un obiect de tip *DataTable* din *DataSet*
- (c) Se închide conexiunea (dacă ea nu a fost explicit deschisă!)

De remarcat că un *DataAdapter* își poate deschide și închide singur conexiunea, dar dacă aceasta a fost deschisă înaintea metodei *Fill()* atunci tot programatorul trebuie să o închidă.

- 3. *Update()* – metodă polimorfică, permițând reflectarea modificărilor efectuate între-un *DataSet*. Pentru a funcționa are nevoie de obiecte de tip comandă adecvate: proprietățile *InsertCommand*, *DeleteCommand* și *UpdateCommand* trebuie să indice către comenzi valide. Returnează de fiecare dată numărul de înregistrări afectate.

### 8.1.2 Proprietăți

- 1. *DeleteCommand*, *InsertCommand*, *SelectCommand*, *UpdateCommand* – de tip *Command*, conțin comenzile ce se execută pentru selectarea sau modificarea datelor în sursa de date. Măcar proprietatea *SelectCommand* trebuie să indice către un obiect valid, pentru a se putea face popularea setului de date.
- 2. *MissingSchemaAction* – de tip enumerare *MissingSchemaAction*, determină ce se face atunci când datele care sunt aduse nu se potrivesc peste schema tablei în care sunt depuse. Poate avea următoarele valori:
  - *Add* - implicit, *DataAdapter* adaugă coloana la schema tablei
  - *AddWithKey* - ca mai sus, dar adaugă și informații relativ la cine este cheia primară
  - *Ignore* - se ignoră lipsa coloanei respective, ceea ce duce la pierdere de date pe *DataSet*
  - *Error* - se generează o excepție de tipul *InvalidOperationException*.

## 8.2 Clasa *DataSet*

Clasa *DataSet* nu mai face parte din biblioteca unui furnizor de date ADO.NET, fiind standard. Ea poate să conțină reprezentări tabelare ale datelor din bază precum și diferite restricții și relații existente. Marele ei

avantaj este faptul că permite lucrul deconectat de la sursa de date, eliminând necesitatea unei conexiuni permanente precum la *DataReader*. În felul acesta, un server de aplicații sau un client oarecare poate apela la serverul de baze de date doar când preia datele sau când se dorește salvarea lor. Funcționează în strânsă legătură cu clasa *DataAdapter* care acționează ca o punte între un *DataSet* și sursa de date. Remarcabil este faptul că un *DataSet* poate face abstracție de sursa de date, procesarea datelor desfășurându-se independent de ea.

Figura 8.1 conține o vedere parțială asupra clasei *DataSet*.

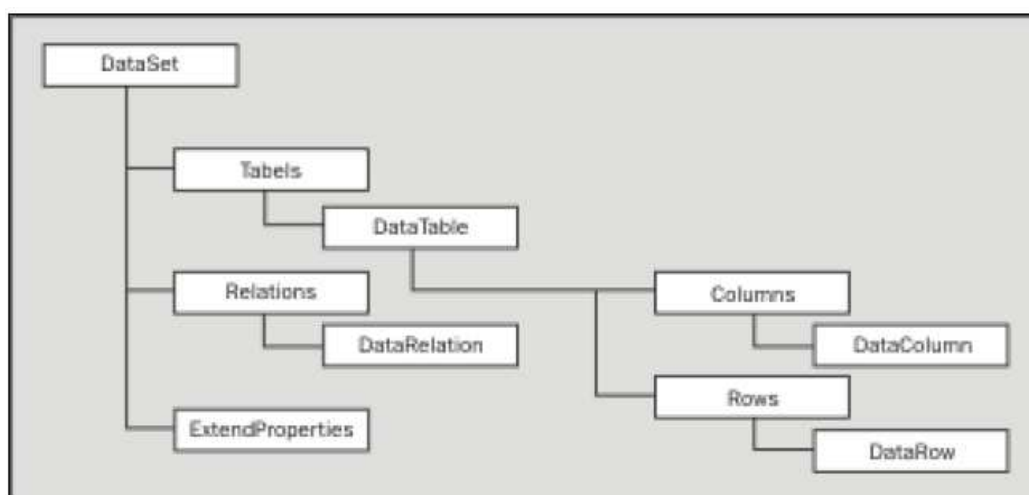


Figura 8.1: Structura unui *DataSet*

### 8.2.1 Conținut

Prezentăm succint conținutul unui *DataSet*:

1. Colecția *Tables* conține 0 sau mai multe obiecte *DataTable*. Fiecare *DataTable* este compusă dintr-o colecție de linii și coloane.
2. Colecția *Relations* conține 0 sau mai multe obiecte de tip *DataRelation*, folosite pentru marcarea legăturilor părinte-copil.
3. Colecția *ExtendedProperties* conține proprietăți definite de utilizator.

### 8.2.2 Clasa *DataTable*

Datele sunt conținute într-un *DataSet* sub forma unor tabele de tip *DataTable*. Aceste obiecte pot fi folosite atât independent, cât și în interiorul unui



*DataSet* ca elemente ale colecției *Tables*. Un *DataTable* conține o colecție *Columns* de coloane, *Rows* de linii și *Constraints* de constrângeri.

### ***DataColumn***

Un obiect *DataColumn* definește numele și tipul unei coloane care face parte sau se adaugă unui obiect *DataTable*. Un obiect de acest tip se obține prin apel de constructor sau pe baza metodei *DataTable.Columns.Add*.

Exemplu:

```
 DataColumn myColumn = new DataColumn("title",
 Type.GetType("System.String"));
```

Definirea unei coloane ca fiind de tip autonumber (în vederea stabilirii ei ca și cheie pe o tabelă) se face astfel:

```
 DataColumn idColumn = new DataColumn("ID",
 Type.GetType("System.Int32"));
 idColumn.AutoIncrement = true;
 idColumn.AutoIncrementSeed = 1;
 idColumn.ReadOnly = true;
```

### ***DataRow***

Un obiect de tip *DataRow* reprezintă o linie dintr-un obiect *DataTable*. Orice obiect *DataTable* conține o proprietate *Rows* ce da acces la colecția de obiecte *DataRow* conținută. Pentru crearea unei linii se poate apela metoda *NewRow* pentru o tabelă a cărei schemă se cunoaște. Mai jos este dată secvența de cod care creează o linie nouă pentru o tabelă și o adaugă acestuia:

```
 DataRow tempRow;
 tempRow = myTable.NewRow();
 tempRow["ID"] = 1;
 tempRow["Name"] = "Book";
 tempRow["Category"] = 1;
 myTable.Rows.Add(tempRow);
```

### **Constrângeri**

Constrângerile sunt folosite pentru a descrie anumite restricții aplicate asupra valorilor din coloane. În ADO.NET există două tipuri de constrângeri: de unicitate și de cheie străină. Toate obiectele de constrângere se află în colecția *Constraints* a unei tabele.

- *UniqueConstraint* - precizează că într-o anumită coloană valorile sunt unice. Încercarea de a seta valori duplicate pe o coloană pentru care s-a precizat restricția duce la aruncarea unei excepții. Este necesară o asemenea coloană în clipa în care se folosește metoda *Find* pentru proprietatea *Rows*: în acest caz trebuie să se specifice o coloană pe care avem unicitate.
- *ForeignKeyConstraint* - specifică acțiunea care se va efectua atunci când se șterge sau modifică valoarea dintr-o anumită coloană. De exemplu se poate decide că dacă se șterge o înregistrare dintr-o tabelă atunci să se șteargă și înregistrările copil. Valorile care se pot seta pentru o asemenea constrângere se specifică în proprietățile *ForeignKeyConstraint.DeleteRule* și *ForeignKeyConstraint.UpdateRule*:
  - *Rule.Cascade* - acțiunea implicită, șterge sau modifică înregistrările afectate
  - *Rule.SetNull* - se setează valoare de **null** pentru înregistrările afectate
  - *Rule.SetDefault* - se setează valoarea implicită definită în bază pentru câmpul respectiv
  - *Rule.None* - nu se execută nimic

Exemplu:

```
ForeignKeyConstraint custOrderFK=new ForeignKeyConstraint
("CustOrderFK",custDS.Tables["CustTable"].Columns["CustomerID"],
 custDS.Tables["OrdersTable"].Columns["CustomerID"]);
custOrderFK.DeleteRule = Rule.None;
//Nu se poate șterge un client care are comenzi facute
custDS.Tables["OrdersTable"].Constraints.Add(custOrderFK);
```

Mai sus s-a declarat o relație de tip cheie străină între două tabele ("CustTable" și "OrdersTable", care fac parte dintr-un DataSet). Restricția se adaugă la tabla copil.

### Stabilirea cheii primare

O cheie primară se definește ca un vector de coloane care se atribuie proprietății *PrimaryKey* a unei tabele (obiect *DataTable*).

```
DataColumn[] pk = new DataColumn[1];
pk[0] = myTable.Columns["ID"];
myTable.PrimaryKey = pk;
```

Proprietatea *Rows* a clasei *DataTable* permite căutarea unei anumite linii din colecția conținută dacă se specifică un obiect sau un array de obiecte folosite pe post de cheie:

```
object key = 17; //cheia dupa care se face cautarea
DataRow line = myTable.Rows.Find(key);
if (line != null)
 //proceseaza linia
```

### 8.2.3 Relații între tabele

Proprietatea *Relations* a unui obiect de tip *DataSet* conține o colecție de obiecte de tip *DataRelation* folosite pentru a figura relațiile de tip părinte-copil între două tabele. Aceste relații se precizează în esență ca niste perechi de array-uri de coloane sau chiar coloane simple din cele două tabele care se relaționează, de exemplu sub forma:

```
myDataSet.Relations.Add(DataColumn, DataColumn);
//sau
myDataSet.Relations.Add(DataColumn[], DataColumn[]);
```

concret:

```
myDataSet.Relations.Add(
 myDataSet.Tables["Customers"].Columns["CustomerID"],
 myDataSet.Tables["Orders"].Columns["CustomerID"]);
```

### 8.2.4 Popularea unui *DataSet*

Deși un obiect *DataSet* se poate popula prin crearea dinamică a obiectelor *DataTable*, cazul cel mai des întâlnit este acela în care se populează prin intermediul unui obiect *DataAdapter*. O dată obținut un asemenea obiect (care conține cel puțin o comandă de tip *SELECT*) se poate apela metoda *Fill()* care primește ca parametru *DataSet*-ul care se umple și opțional numele tabelului care va conține datele:

```
//defineste comanda de selectare din baza de date
String mySqlStmt = "SELECT * FROM Customers";
String myConnectionString = ConfigurationManager.AppSettings["constring"];
//Construiește obiectele de conexiune + comanda SELECT
SqlConnection myConnection = new SqlConnection(myConnectionString);
SqlCommand myCommand = new SqlCommand(mySqlStmt, myConnection);
//Construiește obiectul DataAdapter
```

```
SqlDataAdapter myDataAdapter = new SqlDataAdapter();
//seteaza proprietatea SelectCommand pentru DataAdapter
myDataAdapter.SelectCommand = myCommand;
//construieste obiectul DataSet si il umple cu date
DataSet myDataSet = new DataSet();
myDataAdapter.Fill(myDataSet, "Customers");
```

Datele aduse mai sus sunt depuse într-un obiect de tip *DataTable* din interiorul lui *DataSet*, numit "Customers". Accesul la acest tabel se face prin construcția

```
myDataSet.Tables["Customers"]
```

sau folosind indici întregi (prima tabelă are indicele 0). Același *DataSet* se poate popula în continuare cu alte tabele pe baza aceluiași sau a altor obiecte *DataAdapter*.

### 8.2.5 Clasa *DataTableReader*

Începând cu versiunea 2.0 a lui ADO.NET s-a introdus clasa *DataTableReader* care permite manipularea unui obiect de tip *DataTable* ca și cum ar fi un *DataReader*: într-o manieră *forward-only* și *read-only*. Crearea unui obiect de tip *DataTableReader* se face prin:

```
DataTableReader dtReader = dt.CreateDataReader();
```

iar folosirea lui:

```
while (dtReader.Read())
{
 for (int i = 0; i < dtReader.FieldCount; i++)
 {
 Console.Write("{0} = {1} ",
 dtReader.GetName(i),
 dtReader.GetValue(i).ToString().Trim());
 }
 Console.WriteLine();
}
dtReader.Close();
```

### 8.2.6 Propagarea modificărilor către baza de date

Pentru a propaga modificările efectuate asupra conținutului tabelelor dintr-un *DataSet* către baza de date este nevoie să se definească adecvat obiecte comandă de tip *INSERT*, *UPDATE*, *DELETE*. Pentru cazuri simple se poate folosi clasa *SqlCommandBuilder* care va construi singură aceste comenzi.

#### Clasa *CommandBuilder*

Un obiect de tip *CommandBuilder* (ce provine din furnizorul de date) va analiza comanda *SELECT* care a adus datele în *DataSet* și va construi cele 3 comenzi de update în funcție de aceasta. E nevoie să se satisfacă 2 condiții atunci când se uzează de un astfel de obiect:

1. Trebuie specificată o comandă de tip *SELECT* care să aducă datele dintr-o singură tabelă
2. Trebuie specificată cel puțin cheia primară sau o coloană cu constrângere de unicitate în comanda *SELECT*.

Pentru cea de a doua condiție se poate proceda în felul următor: în comanda *SELECT* se specifică și aducerea cheii, iar pentru obiectul *DataAdapter* care face aducerea din bază se setează proprietatea *MissingSchemaAction* pe valoarea *MissingSchemaAction.AddWithKey* (implicit este doar *Add*).

Fiecare linie modificată din colecția *Rows* a unei tabele va avea modificată valoarea proprietății *RowState* astfel: *DataRowState.Added* pentru o linie nouă adăugată, *DataRowState.Deleted* dacă e ștearsă și *DataRowState.Modified* dacă a fost modificată. Apelul de update pe un *dataReader* va apela comanda necesară pentru fiecare linie care a fost modificată, în funcție de starea ei.

Arătăm mai jos modul de utilizare a clasei *SqlCommandBuilder* pentru adăugarea, modificarea, ștergerea de înregistrări din baza de date.

```
SqlConnection conn = new SqlConnection(
 ConfigurationManager.AppSettings["constring"]);
da = new SqlDataAdapter("SELECT id, name, address FROM
 customers", conn);
da.Fill(ds);
```

```
SqlCommandBuilder cb = new SqlCommandBuilder(da);
//determina liniile care au fost schimbate
DataSet dsChanges = ds.GetChanges();
```

```

if (dsChanges != null)
{
 // modifica baza de date
 da.Update(dsChanges);
 //accepta schimbarile din dataset
 ds.AcceptChanges();
}

```

În clipa în care se creează obiectul *SqlCommandBuilder* automat se vor completa proprietățile *InsertCommand*, *DeleteCommand*, *UpdateCommand* ale *DataAdapter*-ului. Se determină apoi liniile care au fost modificate (prin interogarea stării lor) și se obține un nou *DataSet* care le va conține doar pe acestea. Comanda de *Update* se dă doar pentru acest set de modificări, reducând astfel traficul spre serverul de baze de date.

### Update folosind comenzi SQL

Atunci când interogările de aducere a datelor sunt mai complexe (de exemplu datele sunt aduse din mai multe table, printr-un join) se pot specifica propriile comenzi SQL prin intermediul proprietăților *InsertCommand*, *DeleteCommand*, *UpdateCommand* ale obiectului *DataAdapter*. Pentru fiecare linie dintr-o tabelă care este modificată/adăugată/ștearsă se va apela comanda SQL corespunzătoare. Aceste comenzi pot fi fraze SQL parametrizate sau pot denumi proceduri stocate aflate în bază.

Să presupunem că s-a definit un *DataAdapter* legat la o bază de date. Instrucțiunea de selecție este

```
SELECT CompanyName, Address, Country, CustomerID FROM Customers
```

unde *CustomerID* este cheia. Pentru inserarea unei noi înregistrări s-ar putea scrie codul de mai jos:

```

//da=obiect DataAdapter
da.InsertCommand.CommandText = @"INSERT INTO Customers (
 CompanyName, Address, Country) VALUES
 (@CompanyName, @Address, @Country);
 SELECT CompanyName, Address, Country, CustomerID FROM
 Customers WHERE (CustomerID = @@Identity)";

```

Update-ul efectiv se face prin prima instrucțiune de tip *Update*. Valorile pentru acești parametri se vor da la runtime, de exemplu prin alegerea lor dintr-un tabel. Valoarea pentru cheia *CustomerID* nu s-a specificat, deoarece

(în acest caz) ea este de tip `AutoNumber` (SGBD-ul este cel care face managementul valorilor acestor câmpuri, nu programatorul). `@@Identity` reprezintă id-ul noii înregistrări adăugate în tabelă. Ultima instrucțiune va duce la reactualizarea obiectului *DataSet*, pentru ca acesta să conțină modificarea efectuată (de exemplu ar putea aduce valorile implicite puse pe anumite coloane).

Pentru modificarea conținutului unei linii se poate declara instrucțiunea de *UPDATE* astfel:

```
da.UpdateCommand.CommandText = @"UPDATE Customers SET CompanyName
= @CompanyName, Address = @Address, Country = @Country
WHERE (CustomerID = @ID)";
```

Comanda de update este ca la cazul precedent.

## 8.3 Tranzacții în ADO.NET

O tranzacție este un set de operații care se efectuează fie în întregime, fie deloc. Să presupunem că se dorește trecerea unei anumite sume de bani dintr-un cont în altul. Operația presupune 2 pași:

1. scade suma din primul cont
2. adaugă suma la al doilea cont

Ar fi inadmisibil ca primul pas să reușească iar al doilea să eșueze. Tranzacțiile satisfac niște proprietăți strânse sub numele ACID:

- **atomicitate** - toate operațiile din tranzacție ar trebui să aibă succes sau să eșueze împreună
- **consistență** - tranzacția duce baza de date dintr-o stare stabilă în alta
- **izolare** - nici o tranzacție nu ar trebui să afecteze o alta care rulează în același timp
- **durabilitate** - schimbările care apar în tipul tranzacției sunt permanent stocate pe un mediu.

Sunt trei comenzi care se folosesc în context de tranzacții:

- *BEGIN* - înainte de executarea unei comenzi SQL sub o tranzacție, aceasta trebuie să fie inițializată

- *COMMIT* - se spune că o tranzacție este terminată când toate schimbările cerute sunt trecute în baza de date
- *ROLLBACK* - dacă o parte a tranzacției eșuează, atunci toate operațiile efectuate de la începutul tranzacției vor fi neglijate

Schema de lucru cu tranzacțiile sub ADO.NET este:

1. deschide conexiunea la baza de date
2. începe tranzacția
3. execută comenzi pentru tranzacție
4. dacă tranzacția se poate efectua (nu sunt excepții sau anumite condiții sunt îndeplinite), efectuează *COMMIT*, altfel efectuează *ROLLBACK*
5. închide conexiunea la baza de date

Sub ADO.NET acest lucru s-ar face astfel:

```
SqlConnection myConnection = new SqlConnection(myConnString);
myConnection.Open();
```

```
SqlCommand myCommand1 = myConnection.CreateCommand();
SqlCommand myCommand2 = myConnection.CreateCommand();
SqlTransaction myTrans;
```

```
myTrans = myConnection.BeginTransaction();
//Trebuie asigurate ambele obiecte: conexiune si tranzactie
//unui obiect de tip comanda care va participa la tranzactie
myCommand1.Transaction = myTrans;
myCommand2.Transaction = myTrans;
```

```
try
{
 myCommand1.CommandText = "Insert into Region (RegionID,
 RegionDescription) VALUES (100, 'Description')";
 myCommand1.ExecuteNonQuery();
 myCommand2.CommandText = "Insert into Region (RegionID,
 RegionDescription) VALUES (101, 'Description')";
 myCommand2.ExecuteNonQuery();
 myTrans.Commit();
 Console.WriteLine("Ambele inregistrari au fost scrise.");
}
```



```
}
catch(Exception e)
{
 myTrans.Rollback();
}
finally
{
 myConnection.Close();
}
```

Comanda de *ROLLBACK* se poate executa și în alte situații, de exemplul comanda efectuată depășește stocul disponibil.

## 8.4 Lucrul generic cu furnizori de date

În cele expuse până acum, s-a lucrat cu un furnizor de date specific pentru SQL Server 2005. În general e de dorit să se scrie cod care să funcționeze fără modificări majore pentru orice furnizor de date; mai exact, am prefera să nu fie nevoie de rescrierea sau recompilarea codului. Începând cu versiunea 2.0 a lui ADO.NET se poate face acest lucru ușor, prin intermediul unei clase *DbProviderFactory* (un *Abstract factory*).

Mecanismul se bazează pe faptul că avem următoarele clase de bază pentru tipurile folosite într-un furnizor de date:

- *DbCommand*: clasă de bază abstractă pentru obiectele de tip *Command*
- *DbConnection*: clasă de bază abstractă pentru obiectele de tip *Connection*
- *DbDataAdapter*: clasă de bază abstractă pentru obiectele de tip *DataAdapter*
- *DbDataReader*: clasă de bază abstractă pentru obiectele de tip *DataReader*
- *DbParameter*: clasă de bază abstractă pentru obiectele de tip parametru
- *DbTransaction*: clasă de bază abstractă pentru obiectele de tip tranzacție

Crearea de obiecte specifice (de exemplu obiect *SqlCommand*) se face folosind clase derivate din *DbProviderFactory*; o schiță a acestei clase este:

```
public abstract class DbProviderFactory
{
```

```

...
 public virtual DbCommand CreateCommand();
 public virtual DbCommandBuilder CreateCommandBuilder();
 public virtual DbConnection CreateConnection();
 public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
 public virtual DbDataAdapter CreateDataAdapter();
 public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
 public virtual DbParameter CreateParameter();
}

```

Tot ceea ce trebuie făcut este să se obțină o clasă concretă derivată din *DbProviderFactory* și care la apeluri de tip *Create...* să returneze obiecte concrete, adecvate pentru lucrul cu sursa de date. Concret:

```

static void Main(string[] args)
{
 // Obtine un producator pentru SqlServer
 DbProviderFactory sqlFactory =
 DbProviderFactories.GetFactory("System.Data.SqlClient");
 ...
 // Obtine un producator pentru Oracle
 DbProviderFactory oracleFactory =
 DbProviderFactories.GetFactory("System.Data.OracleClient");
 ...
}

```

Se va evita, firește, codificarea numelui furnizorului de date în cod (precum mai sus) și se vor integra în fișiere de configurare. Aceste șiruri de caractere exemplificate mai sus sunt definite în fișierul *machine.config* din directorul unde s-a făcut instalarea de .NET (%windir%\Microsoft.Net\Framework\v2.0.50727\config).

Exemplu: fișierul de configurare este:

```

<configuration>
<appSettings>
<!-- Provider -->
<add key="provider" value="System.Data.SqlClient" />
<!-- String de conexiune -->
<add key="cnStr" value=
"Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/>
</appSettings>
</configuration>

```

Codul C#:

```
static void Main(string[] args)
{
 string dp = ConfigurationManager.AppSettings["provider"];
 string cnStr = ConfigurationManager.AppSettings["cnStr"];

 DbProviderFactory df = DbProviderFactories.GetFactory(dp);
 DbConnection cn = df.CreateConnection();
 cn.ConnectionString = cnStr;
 cn.Open();

 DbCommand cmd = df.CreateCommand();
 cmd.Connection = cn;
 cmd.CommandText = "Select * From Authors";

 DbDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

 while (dr.Read())
 Console.WriteLine("-> {0}, {1}", dr["au_lname"], dr["au_fname"]);
 dr.Close();
}
```

## 8.5 Tipuri nulabile

Pentru tipurile valoare este mandatorie stabiilirea unei valori; o variabilă de tip valoare nu poate să rețină null. Altfel spus, codul următor va genera eroare de compilare

```
static void Main(string[] args)
{
 bool myBool = null;
 int myInt = null;
}
```

În contextul lucrului cu baze de date este perfect posibil ca rezultatul unei interogări să aducă null pentru un anumit câmp. Pentru a rezolva această incompatibilitate (tipuri cu valoare nenulă în C# care trebuie să poată lucra cu null-urile provenite din bază), s-au introdus tipurile nulabile. Acesta reprezintă un mecanism de extindere a tipurilor de tip valoare astfel încât să suporte și valoarea de nul.

De exemplu, pentru a putea declara o variabilă de tip `int` care să poată avea și valoare nulă se va scrie:

```
int? intNulabil = null;
i=3;
i=null;
```

Nu vom putea defini ca nulabile tipurile referință, deoarece acestea suportă impliciti null:

```
//eroare de compilare
string? s = null
```

Lucrul cu tipurile nulabile se face exact ca și cu tipurile valoare:

```
class DatabaseReader
{
 //campuri nulabile
 public int? numericValue;
 public bool? boolValue = true;

 public int? GetIntFromDatabase()
 { return numericValue; }

 public bool? GetBoolFromDatabase()
 { return boolValue; }
}
```

În contextul tipurilor nulabile s-a introdus operatorul ?? care permite asignarea unei valori pentru o variabilă de tip nulabil dacă valoarea returnată este nulă:

```
static void Main(string[] args)
{
 DatabaseReader dr = new DatabaseReader();

 int? myData = dr.GetIntFromDatabase() ?? 100;
 Console.WriteLine("Value of myData: {0}", myData);
}
```

## Curs 9

# Colecții. Clase generice

### 9.1 Colecții

Un vector reprezintă cel mai simplu tip de colecție. Singura sa deficiență este faptul că trebuie cunoscut dinainte numărul de elemente conținute.

Spațiul de nume *System.Collections* pune la dispoziție un set de clase de tip colecție. O colecție reprezintă un container de elemente de tip *Object* care își gestionează singur necesarul de memorie (crește pe măsură ce se adaugă elemente la el; poate de asemenea să își reducă efectivul de memorie alocat atunci când numărul de elemente efectiv conținute este prea mic).

Exemplu:

```
ArrayList myCollection = new ArrayList();
myCollection.Add(client);
client = (Client)myCollection[0];
```

Remarcăm faptul că este necesară o conversie explicită pentru adresarea unui element din colecție.

Elementele de bază pentru lucrul cu colecțiile sunt un set de interfețe care oferă o mulțime consistentă de metode de lucru. Principalele colecții împreună cu interfețele implementate sunt:

- *ArrayList* : *ICollection*, *IEnumerable*, *ICloneable*
- *SortedList* : *IDictionary*, *ICollection*, *IEnumerable*, *ICloneable*
- *Hashtable* : *IDictionary*, *ICollection*, *IEnumerable*, *ISerializable*, *IDeserializationCallback*, *ICloneable*
- *BitArray* : *ICollection*, *IEnumerable*, *ICloneable*

- *Queue* : *ICollection*, *IEnumerable*, *ICloneable*
- *Stack* : *ICollection*, *IEnumerable*, *ICloneable*
- *CollectionBase* : *ICollection*, *ICollection*, *IEnumerable*
- *DictionaryBase* : *IDictionary*, *ICollection*, *IEnumerable*
- *ReadOnlyCollectionBase* : *ICollection*, *IEnumerable*

### **IEnumerable**

Implementările aceste interfețe permit iterarea peste o colecție de elemente. Unica metodă declarată este metoda *GetEnumerator*:

`IEnumerator GetEnumerator ()`

unde un obiect de tip *IEnumerator* este folosit pentru parcurgerea colecției.

### **ICollection**

Interfața *ICollection* este tipul de bază pentru orice clasă de tip colecție; se derivează din *IEnumerable* și prezintă următoarele proprietăți și metode:

- *Count* - proprietate de tip întreg care returnează numărul de elemente conținute în colecție
- *IsSynchronized* - proprietate logică ce indică dacă colecția este sincronizată (sigură pentru accesarea de către mai multe fire de execuție)
- *SyncRoot* - proprietate care returnează un obiect care poate fi folosit pentru a sincroniza accesul la colecție
- *CopyTo* - metodă care permite copierea conținutului colecției într-un vector

### **ICollection**

Reprezintă o colecție de obiecte care pot fi accesate individual printr-un index.

Proprietățile sunt:

- *IsFixedSize* - returnează o valoare logică indicând dacă lista are o dimensiune fixă

- *IsReadOnly* - returnează o valoare logică indicând dacă lista poate fi doar citită
- *Item* - returnează sau setează elementul de la locația specificată

Metodele sunt:

- *Add* - adaugă un obiect la o listă
- *Clear* - golește lista
- *Contains* - determină dacă colecția conține o anumită valoare
- *IndexOf* - determină poziția în listă a unei anumite valori
- *Insert* - inserează o valoare la o anumită poziție
- *Remove* - șterge prima apariție a unui obiect din listă
- *RemoveAt* - șterge un obiect aflat la o anumită locație

## IDictionary

Interfața *IDictionary* reprezintă o colecție de perechi (cheie, valoare). Permite indexarea unei colecții de elemente după altceva decât indici întregi. Fiecare pereche trebuie să aibă o cheie unică.

Proprietățile sunt:

- *IsFixedSize*, *IsReadOnly* - returnează o valoare care precizează dacă colecția este cu dimensiune maximă fixată, respectiv doar citibilă
- *Item* - returnează un element având o cheie specificată
- *Keys* - returnează o colecție de obiecte conținând cheile din dicționar
- *Values* - returnează un obiect de tip colecție care conține toate valorile din dicționar

Metodele sunt:

- *Add* - adaugă o pereche (cheie, valoare) la dicționar; dacă cheia există deja, se va face suprascrierea valorii asociate
- *Clear* - se șterge conținutul unui dicționar
- *Contains* - determină dacă dicționarul conține un element cu cheia specificată

- *GetEnumerator* - returnează un obiect de tipul *IDictionaryEnumerator* asociat
- *Remove* - șterge elementul din dicționar având cheia specificată

### 9.1.1 Iteratori pentru colecții

Colecțiile (atât cele de tip listă, cât și cele dicționar) moștenesc interfața *IEnumerable* care permite construirea unui obiect de tip enumerator instanță a lui *IEnumerable*:

```
interface IEnumerator {
 object Current {get;}
 bool MoveNext();
 void Reset();
}
```

Remarcăm că un asemenea iterator este de tip forward-only. Proprietatea *Current* returnează elementul curent al iterării. Metoda *MoveNext* avansează la următorul element al colecției, returnând *true* dacă acest lucru s-a putut face și *false* în caz contrar; trebuie să fie apelată cel puțin o dată înaintea accesării componentelor colecției. Metoda *Reset* reinițializează iteratorul mutând poziția curentă înaintea primului obiect al colecției.

Pentru fiecare clasă de tip colecție, enumeratorul este implementat ca o clasă internă. Returnarea unui enumerator se face prin apelul metodei *GetEnumerator*.

Exemplu: se va apela în mod explicit metoda de returnare a iteratorului și mai departe acesta se folosește pentru afișarea elementelor.

```
ArrayList list = new ArrayList();
list.Add("One");
list.Add("Two");
list.Add("Three");

IEnumerator e = list.GetEnumerator();
while(e.MoveNext())
{
 Console.WriteLine(e.Current);
}
```

Apelul unui iterator este mecanismul de bază pentru instrucțiunea *foreach*, care debutează prin a apela intern metoda *GetEnumerator* iar trecerea la elementul următor se face cu metoda *MoveNext*. Altfel spus, exemplul de mai sus este echivalent cu:



```
ArrayList list = new ArrayList();
list.Add("One");
list.Add("Two");
list.Add("Three");

foreach(String s in list)
{
 Console.WriteLine(s);
}
```

### 9.1.2 colecții de tip listă

Colecțiile de tip listă sunt: *ArrayList*, *BitArray*, *Stack*, *Queue* și *CollectionBase*.

#### ArrayList

Este o clasă concretă care stochează o colecție de elemente sub forma unui vector auto-redimensionabil. Suportă mai mulți cititori concurenți și poate fi accesat exact ca un vector:

```
ArrayList list = new ArrayList();
list.Add(...);
Console.WriteLine(list[0]);
list[0] = "abc";
```

#### BitArray

Acest tip de colecție gestionează un vector de elemente binare reprezentate ca booleeni, unde *true* reprezintă 1 iar *false* 0. Cele mai importante metode sunt:

- *And*, *Or*, *Xor* - produce un nou *BitArray* care conține rezultatul aplicării operanzilor respectivi pe elementele din colecția curentă și altă colecție dată ca argument. Dacă cele 2 colecții nu au același număr de elemente, se aruncă excepție
- *Not* - returnează un obiect de tip *BitArray* care conține valorile negate din colecția curentă

#### Stack

*Stack* reprezintă o colecție ce permite lucrul conform principiului *LIFO* - *Last In, First Out*.

## Queue

Clasa *Queue* este nou apărută în versiunea 2.0 ce permite implementarea politicii *FIFO* - *First In, First Out*.

## CollectionBase

Clasa *CollectionBase* reprezintă o clasă abstractă, bază pentru o colecție puternic tipizată. Programatorii sunt încurajați să deriveze această clasă decât să creeze una proprie de la zero.

### 9.1.3 Colecții de tip dicționar

Colecțiile de tip dicționar (*SortedList*, *Hashtable*, *DictionaryBase*) conțin obiecte care se manipulează prin intermediul cheii asociate (care poate fi altceva decât un indice numeric). Toate extind interfața *IDictionary*, iar ca enumeratorul este de tip *IDictionaryEnumerator*:

```
interface IDictionaryEnumerator : IEnumerator {
 DictionaryEntry Entry {get;}
 object Key {get;}
 object Value {get;}
}
```

unde *DictionaryEntry* este definit ca:

```
struct DictionaryEntry {
 public DictionaryEntry(object key, object value) { ... }
 public object Key {get; set;}
 public object Value {get; set;}
 ...
}
```

Invocarea enumeratorului se poate face fie explicit:

```
Hashtable htable = new Hashtable();
htable.Add("A", "Chapter I");
htable.Add("B", "Chapter II");
htable.Add("App", "Appendix");
IDictionaryEnumerator e = htable.GetEnumerator();
for (; e.MoveNext() ;)
 Console.WriteLine(e.Key);
```

fie implicit:

```
foreach (DictionaryEntry s in htable)
 Console.WriteLine(s.Key);
```

### Hashtable

Reprezintă o colecție de perechi de tip (cheie, valoare) care este organizată pe baza codului de dispersie (hashing) al cheii. O cheie nu poate să fie nulă. Obiectele folosite pe post de chei trebuie să suprascrie metodele *Object.GetHashCode* și *Object.Equals*. Obiectele folosite pe post de cheie trebuie să fie imuabile (să nu suporte schimbări de stare care să altereze valorile returnate de cele 2 metode spuse anterior).

### SortedList

Reprezintă o colecție de perechi de tip (cheie, valoare) care sunt sortate după cheie și se pot accesa după cheie sau după index.

### DictionaryBase

Reprezintă o clasă de bază abstractă pentru implementarea unui dicționar utilizator puternic tipizat (valorile să nu fie văzute ca *object*, ci ca tip specificat de programator).

## 9.2 Crearea unei colecții

Vom exemplifica în această secțiune modul în care se definește o colecție ce poate fi iterată. Sunt prezentate 2 variante: specifice versiunilor 1.1 și respectiv 2.0 ale platformei .NET. În ambele cazuri clasa de tip colecție va implementa interfața *IEnumerable*, dar va diferi modul de implementare.

### 9.2.1 Colecție iterabilă (stil vechi)

```
using System;
using System.Collections;
class MyCollection : IEnumerable
{
 private int[] continut = {1, 2, 3};
 public IEnumerator GetEnumerator()
 {
 return new MyEnumerator(this);
 }
}
```

```

private class MyEnumerator : IEnumerator
{
 private MyCollection mc;
 private int index = -1;

 public MyEnumerator(MyCollection mc)
 {
 this.mc = mc;
 }

 public object Current
 {
 get
 {
 if (index < 0 || index >= mc.continut.Length)
 {
 return null;
 }
 else return mc.continut[index];
 }
 }

 public bool MoveNext()
 {
 index++;
 return index < mc.continut.Length;
 }

 public void Reset()
 {
 index = -1;
 }
}

```

Remarcăm că clasa interioară primește prin constructor o referință la obiectul de tip colecție, deoarece orice clasă interioară în C# este automat și statică (neavând acces la membrii nestatici ai clasei).

Demonstrația pentru iterarea clasei este:

```

class TestIterator

```

```
{
 static void Main()
 {
 MyCollection col = new MyCollection();
 foreach(int i in col)
 {
 Console.WriteLine(s);
 }
 }
}
```

Instrucțiunea *foreach* va apela inițial metoda *GetEnumerator* pentru a obține obiectul de iterare și apoi pentru acest obiect se va apela metoda *MoveNext* la fiecare iterație. Dacă se returnează *true* atunci se apelează automat și metoda *Current* pentru obținerea elementului curent din colecție; dacă se returnează *false* atunci execuția lui *foreach* se termină.

Implementarea de mai sus permite folosirea simultană a mai multor iteratori, cu păstrarea stării specifice.

Defectele majore ale acestei implementări sunt:

1. Complexitatea codului (numărul mare de linii). Deși ușor de înțeles și general acceptată (fiind de fapt socotit un design pattern), abordarea presupune scrierea multor linii de cod, motiv pentru care programatorii evita aceasta facilitate, preferând mecanisme alternative precum indexatorii.
2. Datorită semnăturii metodei *Current* se returnează de fiecare dată un *Object*, pentru care se face fie boxing și unboxing (dacă în colecție avem tip valoare - cazul de mai sus), fie downcasting (de la *Object* la tipul declarat în prima parte a lui *foreach*, dacă în colecție avem tip referință). În primul caz resursele suplimentare de memorie și ciclul procesor vor afecta negativ performanța aplicației iar în al doilea caz apare o conversie explicită care dăunează performanței globale. Modalitatea de evitare a acestei probleme este ca să nu se implementeze interfețele *IEnumerator* și *IEnumerable*, ci scriind metoda *Current* astfel încât să returneze direct tipul de date necesar (*int* în cazul nostru). Acest lucru duce însă la expunerea claselor interioare (văzute ca niște clase auxiliare), ceea ce încalcă principiul încapsularii. În plus, cantitatea de cod rămâne aceeași.

Pentru prima problemă vom da varianta de mai jos. Pentru cea de a doua, rezolvarea se dă sub forma claselor generice.

### 9.2.2 Colecție iterabilă (stil nou)

Începând cu C# 2.0 se poate defini un iterator mult mai simplu. Pentru aceasta se folosește instrucțiunea *yield*. *yield* este folosită într-un bloc de iterare pentru a semnală valoarea ce urmează a fi returnată sau oprirea iterării. Are formele:

```
yield return expression;
yield break;
```

În prima formă se precizează care va fi valoarea returnată; în cea de-a doua se precizează oprirea iterării (sfârșitul secvenței de elemente de returnat).

Pentru exemplificare, prezentăm o metodă al cărei rezultat este folosit pentru iterare. Valorile returnate de această metodă sunt pătratele numerelor de la 1 la valoarea argumentului

```
using System;
using System.Collections;
using System.Text;

namespace TestCollection
{
 class Program
 {
 static IEnumerable Squares(int number)
 {
 for (int i = 1; i <= number; i++)
 {
 yield return i*i;
 }
 }

 static void Main(string[] args)
 {
 foreach (int iterate in Squares(10))
 {
 Console.WriteLine(iterate.ToString());
 }
 }
 }
}
```

Remarcăm că are loc următorul efect: la fiecare iterație se returnează următoarea valoare din colecție (colecția este definită de metoda *Squares*). Astfel, se creează impresia că la fiecare iterație din metoda *Main* se reia execuția din metoda *Squares* de unde a rămas la apelul precedent; acest mecanism este diferit de cel al rutinelor (metodelor) întâlnite până acum, purtând numele de *corutină*. De fapt, compilatorul va genera automat o implementare de metodă de tip *IEnumerable* (precum am făcut manual în secțiunea 9.2.1), permițându-se astfel programatorului să se concentreze pe designul metodei și mai puțin pe stufoasele detalii interne.

Clasa *MyCollection* de mai sus s-ar rescrie astfel:

```
class MyCollection : IEnumerable
{
 private int[] continut = { 1, 2, 3 };
 public IEnumerator GetEnumerator()
 {
 for(int i=0; i<continut.Length; i++)
 {
 yield return continut[i];
 }
 }
}
```

Pentru a demonstra utilitatea acestui tip de implementare, mai jos dăm rezolvarea pentru următoarea problemă: plecându-se de la un arbore binar să se scrie iteratorii pentru parcurgerea în inordine și preordine. Nu vom prezenta construirea efectivă a arborelui, aceasta fiind o problemă separată. Practic, se va implementa în mod recursiv o iterație peste arbore.

Pentru început, definiția tipului nod:

```
using System;
namespace TestIterTree
{
 class TreeNode<T>
 {
 private T value;
 private TreeNode<T> left, right;

 public T Value
 {
 get
 {
```

```

 return value;
 }
 set
 {
 this.value = value;
 }
}

public TreeNode<T> Left
{
 get
 {
 return left;
 }
 set
 {
 left = value;
 }
}

public TreeNode<T> Right
{
 get
 {
 return right;
 }
 set
 {
 this.right = value;
 }
}
}
}

```

Urmează definirea arborelui și a celor doi iteratori:

```

using System;
using System.Collections.Generic;

namespace TestIterTree
{
 class Tree<T>

```



```
{
 private TreeNode<T> root;

 #region Populate tree with values
 public void AddValues(params T[] value)
 {
 Array.ForEach(value, Add);
 }
 #endregion

 #region Various traversal techniques
 public IEnumerable<T> InOrder()
 {
 return InOrderTraversal(root);
 }

 public IEnumerable<T> PreOrder()
 {
 return PreOrder(root);
 }
 #endregion

 #region Private helper methods
 private IEnumerable<T> InOrderTraversal(TreeNode<T> node)
 {
 if (node.Left != null)
 {
 foreach (T value in InOrderTraversal(node.Left))
 {
 yield return value;
 }
 }
 yield return node.Value;
 if (node.Right != null)
 {
 foreach (T value in InOrderTraversal(node.Right))
 {
 yield return value;
 }
 }
 }
}
```

```

private IEnumerable<T> PreOrder(TreeNode<T> root)
{
 yield return root.Value;
 if (root.Left != null)
 {
 yield return root.Value;
 }
 if (root.Left != null)
 {
 foreach (T value in PreOrder(root.Left))
 {
 yield return value;
 }
 }
 if (root.Right != null)
 {
 foreach (T value in PreOrder(root.Right))
 {
 yield return value;
 }
 }
}

private void Add(T value)
{
 //Implements adding a value to the tree
}
#endregion
}
}

```

Implementarea de mai sus s-a făcut conform definițiilor recursive pentru cele două tipuri de parcurgeri (al treilea tip de parcurgere se implementează analog). Invităm cititorul să compare aceste implementări cu cele iterative clasice din teoria structurilor de date. Pe lângă timpul scurt de implementare, se câștigă în claritate și ușurință în exploatare.

## 9.3 Clase generice

Vom prezenta în cele ce urmează suportul .NET 2.0 pentru clase și metode generice; acestea sunt blocuri de cod parametrizate care permit scrierea unui cod general, ce poate fi ulterior adaptat automat la cerințele specifice ale programatorului.

### 9.3.1 Metode generice

Să presupunem că dorim să scriem o metodă care să realizeze interschimbarea valorilor a două variabile. Variantele sunt:

1. scrierea unei metode pentru fiecare tip al variabilelor: neelegant, cod mult
2. scrierea unei metode care să folosească un *Object* pe post de variabilă auxiliară; dacă se face apelul pentru 2 variabile de tip șir de caractere, apare eroarea “Cannot convert from ‘ref string’ to ‘ref object’”. În plus, antetul metodei ar permite apel pentru un parametru de tip *string* și celălalt de tip *int*, ceea ce nu ar trebui să fie admis la compilare.

Singurul mod adecvat de rezolvare a problemei este folosirea unei metode generice, ca mai jos:

```
void Swap<T>(ref T a, ref T b)
{
 T aux;
 aux = a;
 a = b;
 b = aux;
}
```

Apelul acestei metode se face astfel:

```
int x = 3, y=4;
Swap<int>(ref x, ref y); //nu apare boxing/unboxing
string a="a", b="b";
Swap<string>(ref a, ref b);
```

Remarcăm că apelul se face specificând tipul efectiv pentru *T*. Această specificație poate fi omisă dacă compilatorul poate deduce singur care este tipul efectiv *T*:

```
bool b1=true, b2=false;
Swap(ref b1, ref b2);
```

Tipul generic *T* poate fi folosit și ca tip de retur.

### 9.3.2 Tipuri generice

Mecanismul de genericitate poate fi extins la clase și structuri. Dăm mai jos exemplu care modelează noțiunea de punct într-un spațiu bidimensional. Genericitatea provine din faptul că coordonatele pot fi de tip întreg sau fracționare.

```
struct Point<T>
{
 private T xPos;
 private T yPos;

 public Point(T xPos, T yPos)
 {
 this.xPos = xPos;
 this.yPos = yPos;
 }

 public T X
 {
 get
 {
 return xPos;
 }
 set
 {
 xPos = value;
 }
 }

 public T Y
 {
 get
 {
 return yPos;
 }
 set
 {
 yPos = value;
 }
 }
}
```

```

public override string ToString()
{
 return string.Format("{0}, {1}", xPos.ToString(),
 yPos.ToString());
}

public void Reset()
{
 xPos = default(T);
 yPos = default(T);
}
}

```

Utilizarea efectivă ar putea fi:

```

Point<int> p = new Point(10, 10);
Point<double> q = new Point(1.2, 3.4);

```

Observăm că:

- Metodele, deși cu caracter generic, nu se mai specifică drept generice (nu se mai folosesc simbolurile < și >), acest lucru fiind implicit
- folosim o supraîncărcare a cuvântului cheie *default* pentru a aduce câmpurile la valorile implicite ale tipului respectiv: 0 pentru numerice, *false* pentru boolean, *null* pentru tipuri referință.

Mai adăugăm faptul că o clasă poate avea mai mult de un tip generic drept parametru, exemplele clasice fiind colecțiile generice de tip dicționar pentru care se specifică tipul cheii și al valorilor conținute.

### 9.3.3 Constrângeri asupra parametrilor de genericitate

Pentru structura de mai sus este posibil să se folosească o instanțiere de tipul:

```
Point<StringBuilder> r = null;
```

ceea ce este aberant din punct de vedere semantic. Am dori să putem face restricționarea tipului parametrilor generici. Un asemenea mecanism există și permite 5 tipuri de restricții:

<i>where T:struct</i>	<i>T</i> trebuie să fie tip derivat din <i>System.Value</i>
<i>where T:class</i>	<i>T</i> trebuie să nu fie derivat din <i>System.Value</i>
<i>where T:new()</i>	<i>T</i> trebuie să aibe un constructor implicit (fără parametri)
<i>where T:NameOfBaseClass</i>	<i>T</i> trebuie să fie derivat (direct sau nu) din <i>NameOfBaseClass</i>
<i>where T:NameOfInterface</i>	<i>T</i> trebuie să implementeze interfața <i>NameOfInterface</i>

Exemple:

- `class MyGenericClass<T> where T:new()` specifică faptul că parametrul *T* trebuie să fie un tip cu constructor implicit
- `class MyGenericClass<T> where T:class, IDrawable, new()` specifică faptul că parametrul *T* trebuie să fie de tip referință, să implementeze *IDrawable* și să posede constructor implicit
- `class MyGenericClass<T>:MyBase, ICloneable where T:struct` descrie o clasă care este derivată din *MyBase*, implementează *ICloneable* iar parametrul *T* este de tip valoare (structură sau enumerare).

Clasele generice pot fi de asemenea clase de bază pentru tipuri (generice sau nu):

```
class MyList<T>...
class MyStringList : MyList<String>...
```

### 9.3.4 Interfețe și delegați generici

Interfețele și delegații pot fi declarate ca fiind generice; deși nu prezintă cerințe sau particularități față de ceea ce s-a spus mai sus, le evidențiem astfel deoarece gradul înalt de abstractizare le face deosebit de utile în modelarea unui sistem soft complex.

```
interface IMyFeature<T>
{
 T MyService(T param1, T param2);
}
```

respectiv:

```
delegate void MyGenericDelegate<T>(T arg);
```

## 9.4 Colecții generice

### 9.4.1 Probleme cu colecțiile de obiecte

Colecțiile, așa cum au fost ele prezentate în secțiunea 9.1 sunt utile, dar au câteva puncte slabe.

1. să presupunem că pornim cu o listă de tip *ArrayList* la care adăugăm elemente de tip întreg:

```
ArrayList al = new ArrayList();
al.Add(1);
al.Add(2);
int x = (int)al[0];
```

Secvența este corectă din punct de vedere sintactic, dar la rulare solicită folosirea mecanismului de boxing și unboxing. Deși pentru colecții mici acest lucru nu are efecte sesizabile, pentru un număr mare de adăugări sau accesări ale elementelor din listă avem un impact negativ ce trebuie luat în calcul. Am prefera ca tipurile colecție să suporte lucrul cu tipuri valoare fără costul suplimentar introdus de boxing/unboxing.

2. problema tipului efectiv stocat în colecție: să presupunem că într-o listă adăugăm:

```
al.Add(new Dog("Miki"));
al.Add(new Dog("Gogu"));
al.Add(new Matrix(3, 5));
Dog dog = (Dog)al[2];
```

Secvența de sus este corectă din punct de vedere sintactic, dar la rulare va determina aruncarea unei excepții de tipul *InvalidCastException*. E de dorit ca la compilare să se poată semnala această greșeală.

### 9.4.2 Colecții generice

Clasele generice împreună cu colecțiile au fost combinate în biblioteca .NET Framework, ducând la apariția unui nou spațiu de nume, conținut în *System.Collections.Generic*. Acesta conține tipurile: *ICollection<T>*, *IComparer<T>*, *IDictionary<K, V>*, *IEnumerable<T>*, *IEnumerator<T>*, *IList<T>*, *Queue<T>*, *Stack<T>*, *LinkedList<T>*, *List<T>*.

Exemplu de utilizare:

```
List<int> myInts = new List<int>();
myInts.Add(1);
myInts.Add(2);
//myInts.Add(new Complex()); //eroare de compilare
```

Deși în secvența de mai sus tipul listei este *int*, nu se apelează la boxing/unboxing, deoarece lista este compusă din elemente de tip întreg și nu din obiecte de tip *Object*.



# Curs 10

## ASP.NET

ASP.NET (actualmente versiunea 2.0) permite crearea de aplicații Web folosind platforma .NET Framework; este una din cele 3 tipuri de aplicații creabile sub această platformă, alături de Windows Forms și Web Services. Este destinată să înlocuiască vechiul ASP (Active Server Pages), fiind obiectual și integrat cu toate elementele constitutive ale lui .NET Framework: CLR, BCL, CTS, FCL, etc. Se poate rula pe servere de aplicații IIS (versiuni 5, 5.1, 6). Rezultatele cele mai bune se obțin pe IIS 6.0 (integrat în Windows 2003 Server).

Enunțăm mai jos o listă neexhaustivă a trăsăturilor ASP.NET:

- unul din marile avantaje ale lui ASP.NET față de ASP este faptul că codul este păstrat compilat (sub formă de cod .NET PE, executat de către CLR) în loc ca pagina să fie interpretată la fiecare rulare;
- modul de realizare este orientat pe obiecte, o aplicație fiind compusă din două părți: pagina afișată în browser și pagina de “code-behind”, care este o clasă; acest code-behind are posibilitatea de accesare a bazelor de date sau chiar a codului unmanaged (COM-uri și DLL-uri anterior existente); în plus, se realizează separarea părții de interfață utilizator de ceea ce este executat pe server;
- pune la dispoziție un mod de tratare a erorilor unificat (tratarea de excepții) și folosește controale Web predefinite care se pot executa pe server sau pe client;
- are un mecanism încorporat care permite păstrarea stării paginii (chiar dacă protocolul de comunicare HTTP este fără stare, stateless);
- ASP.NET permite includerea de mai multe funcționalități pe controalele ASP.NET iar managementul stării lor este mult îmbunătățit

- permite programarea bazată pe evenimente, cu diferențe minime față de programarea aplicațiilor windows;
- este senzitiv la browserul pentru care se face trimiterea codului HTML, alegând automat codul optimizat pentru o colecție de browsere
- erorile grave apărute în aplicație ASP.NET nu vor duce la oprirea serverului Web (dar pot duce la imposibilitatea de a mai executa pagini ASP.NET până la repornirea serverului)
- permite scrierea codului care se va executa pe server în orice limbaj .NET: C#, VB.NET, etc

## 10.1 Anatomia unei pagini ASP.NET

O pagină ASP.NET se poate crea într-o multitudine de feluri, de exemplu prin Notepad, Visual Studio.NET sau Web Developer Express Edition. Se poate crea o pagină simplă pe baza următorului conținut:

```
<html>
 <head>
 </head>
 <body>
 Hello World
 </body>
</html>
```

salvat sub numele de Default.aspx, creată în cadrul unui proiect VS.NET 2005. Pagina se încarcă în browser folosind adresa <http://localhost:1298/HelloWorld/Default.aspx>, numărul 1298 fiind un port liber, alocat aleator de către mediul de dezvoltare.

### 10.1.1 Adăugarea unui control Web

Într-o pagină Web se poate adăuga un control pe pagină care să fie generat de către serverul IIS în mod adecvat și care poate fi accesat programatic prin cod. Astfel, în pagina de mai sus se adaugă astfel încât să devină:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
```

```

<head runat="server">
 <title>Hello World page</title>
</head>
<body>
 <form id="form1" runat="server">
 <div>
 <asp:Label ID="lblHelloWorld" runat="server"
 Text="Hello World"></asp:Label>
 </div>
 </form>
</body>
</html>

```

Dacă se mai încarcă o dată pagina, se va genera următorul cod HTML pentru browser (vizibil prin opțiunea View Source):

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR
<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>
Hello World page
</title></head>
<body>
 <form name="form1" method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/wEPDwUJODExMDE5NzY
</div>

 <div>
 Hello World
 </div>
</form>
</body>
</html>

```

În sursa s-a încorporat un control Web `<asp:Label>` care prezintă proprietatea *Text* cu textul care va fi afișat în pagină. Atributul *runat* are ca valoare cuvântul *server* (singura opțiune posibilă) care spune că respectivul control se va procesa pe server și nu pe client. După cum se vede în sursa HTML generată, acest control va avea ca rezultat un tag de tip *span*. De remarcat că controlul Web a fost scris în interiorul unui formular; acest lucru este mandatoriu pentru toate controalele Web sau HTML.

### 10.1.2 Adăugarea scriptului *inline*

Vom adăuga la pagina anterioară un buton a cărui apăsare va duce la schimbarea textului etichetei anterioare; această schimbare va fi făcută pe server prin apelul unei metode scrise în C#:

```
<html>
 <head>
 </head>
 <body>
 <form method="post" runat="server">
 <asp:Label id=lblHelloWorld text="Hello World" runat="Server"/>

 <asp:Button onclick="ClickedIt" text="Submit" runat="Server" />
 </form>
 </body>
 <script Language="C#" runat="server">
 void ClickedIt(object sender, System.EventArgs e)
 {
 lblHelloWorld.Text = "Text schimbat";
 }
 </script>
</html>
```

La apăsarea butonului se va face submiterea formularului către server, spre aceeași pagină HelloWorld.aspx (lucru care se poate verifica din inspectarea codului HTML generat). Pentru a se putea accesa controlul de tip Label pe partea de server, trebuie ca acestuia să i se asigneze un ID (lblHelloWorld în cazul nostru). Pentru controlul Button (reprezentat de tag-ul <asp:Button\>) s-au asigurat textul și metoda care se va apela pe server (ClickedIt). Codul pentru ClickedIt este scris inline în aceeași pagină ASP.NET (pentru simplitate, dar vom vedea că această metodă are o variantă mai bună). Remarcăm că dacă s-a apăsă pe buton apoi se face refresh pe browser se va retransmite “Text schimbat”, deoarece serverul menține starea controalelor.

Cea mai importantă este observația că la codul HTML generat pe client s-a adăugat un element input de tip hidden cu numele *\_\_VIEWSTATE*. Acesta este mecanismul prin care ASP.NET menține starea unei pagini de la o trimiteră la alta către server. Prima dată când se cere pagina, ASP.NET va codifica valoarea tuturor controalelor și le va stoca în acest input ascuns. La fiecare cerere următoare a paginii acest input va fi folosit pentru a reinițializa controalele Web și HTML. În felul acesta este menținută starea unor con-

troale (de exemplu conținutul unui input de tip text sau de tip select, pentru care s-a făcut alegerea unei opțiuni).

### 10.1.3 Code-behind

Se recomandă evitarea scrierii codului inline, deoarece asta duce la amestecarea părții de design al paginii cu procesarea evenimentelor; în plus, este posibil ca procesarea să fie extrem de consistentă iar codul rezultat devine greu de urmărit și menținut.

ASP.NET are o modalitate extrem de simplă pentru separarea părții de interfață utilizator de cea de cod propriu-zis. E utilă separarea între cele două aspecte, deoarece de partea de interfață se poate ocupa un web-designer, iar de cea programativă cineva cu cunoștințe de .NET. Codul executat se va scrie într-un fișier cu extensia `aspx.cs` sau `cs` (dacă e vorba de cod C#) care va fi compilat sub forma unui fișier `dll` depus în directorul `bin` al aplicației Web.

Pentru a se introduce partea de code-behind pentru pagina anterioară, se șterge scriptul inline și se modifică astfel:

```
<%@ Page Inherits="Test1.HelloWorldCB" %>
<html>
 <head>
 </head>
 <body>
 <form method="post" runat="server" >
 <asp:Label id="lblHelloWorld" text="Hello World" runat="Server" />

 <asp:Button onclick="ClickedIt" text="Submit" runat="Server" />
 </form>
 </body>
</html>
```

Pe primul rând s-a scris o directivă ASP.NET (ele pot fi scrise oriunde, dar se obișnuiește trecerea lor la începutul documentului). Directiva `Page` precizează în acest caz care este clasa din care se derivează pagina curentă. Clasa de bază se poate declara așa:

```
namespace Test1
{
 using System;
 using System.Web;
 using System.Web.UI;
```

```
using System.Web.UI.WebControls;
public partial class HelloWorldCB: System.Web.UI.Page
{
 protected Label lblHelloWorld;
 protected void ClickedIt(object sender, System.EventArgs e)
 {
 lblHelloWorld.Text = "Text schimbat" ;
 }
}
```

Pagina aspx rezultată va moșteni clasa Test1.HelloWorldCB.

## 10.2 Clasa *Page*

Clasa *Page* este moștenită direct sau indirect de orice pagină ASP.NET; ea este responsabilă de asigurarea unei funcționalități de bază pentru procesarea de pagini pe server.

Ciclul de viață a unei pagin ASP.NET este următorul:

1. Pagina este inițializată. Aceasta include crearea instanțelor controalelor și setarea event-handlerelor.
2. Partea de viewstate este procesată, populându-se controalele cu date.
3. Evenimentul de Load este apelat. În event-handlerul pentru acest eveniment se începe programarea logicii paginii.
4. Event-handlerele pentru controale sunt apelate (apăsări de butoane, schimbarea selecției curente dintr-un dropdown list, etc)
5. Se salvează viewstate
6. Pagina este tradusă în HTML (randare)

Dăm mai jos cele mai importante evenimente din clasa *Page*.

### 10.2.1 Evenimentul *Init*

Acest eveniment este primul apelat în ciclul de viață al unei pagini. În interiorul event-handlerului se creează toate controalele de pe pagină și de asemenea se face asignează event-handlerele pentru acestea.

### 10.2.2 Evenimentul *Load*

Se apelează după *Init*. Are acces la partea de viewstate a paginii (inaccesibilă pentru *Init*). Această metodă poate decide dacă este cerută de către client pentru prima dată sau este rezultatul unui postback, pe baza proprietății boolene *IsPostBack*). Schița unei metode *Load* ar fi:

```
private void Page_Load(object sender, System.EventArgs e)
{
 //cod care se executa la fiecare cerere
 if(!IsPostBack)
 {
 //cod care trebuie executat doar la prima cerere a paginii
 //ex: aducerea unor date din baza si popularea unor controale
 }
 else
 {
 //cod care se executa doar daca s-a facut postback
 //ex: verificarea unor intrari de forma
 }
}
```

### 10.2.3 Evenimentul *Unload*

Este apelat la descărcarea paginii din memoria serverului. Aici se dealocă resurse precum conexiuni la baza de date.

## 10.3 Crearea unei pagini ASP.NET folosind Visual Studio.NET

Deși se poate scrie ASP.NET și “de mână”, este de preferat folosirea VS.NET pentru că permite crearea simplă a fișierelor aspx și a celor de code-behind.

Fișierul Web.config creat automat de către mediu conține setări (de securitate, legate de sesiune) specifice aplicației.

De asemenea s-a creat și un web form (alt nume pentru pagină ASP.NET) numit implicit Webform1.aspx; din partea de Solution Explorer se poate modifica numele lui cu unul adecvat, de exemplu HelloWorld.aspx (se folosește această denumire în cele ce urmează).

Pe prima linie a fișierului HelloWorld.aspx apare:

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

Pentru directiva Page s-au specificat următoarele atribute:

- *language* cu valoarea *c#*; este limbajul în care se scrie în toate blocurile de cod de tip server;
- *Codebehind* specifică numele fișierului sursă; acesta se creează automat de către VS.NET;
- *Inherits* setează clasa care este moștenită de forma web; această clasă trebuie să fie derivată direct sau indirect din clasa *System.Web.UI.Page*.

Se trage de pe toolbox un control Label pe forma ASP.NET, iar în partea de proprietăți se setează pentru ID valoarea *lblHelloWorld*, iar pentru text valoarea “Hello World.” Codul generat de designer va fi:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inher
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
 <title>Hello World page</title>
</head>
<body>
 <form id="form1" runat="server">
 <div>
 <asp:Label ID="lblHelloWorld" runat="server" Text="Hello World"></asp
 </div>
 </form>
</body>
</html>
```

Se trage apoi un control de tip buton pe formă pentru care ID-ul implicit este *Button1* iar textul va fi setat la “Buton”; pe partea de HTML a formularului codul va fi:

```
<form id="form1" runat="server">
 <div>
 <asp:Label ID="lblHelloWorld" runat="server" Text="Hello World"></asp
 <asp:Button ID="Button1" runat="server" Text="Buton" /></div>
</form>
```



Dacă în toolbox, la partea de evenimente atașate butonului se dă dublu click pe evenimentul *Click* în partea de code-behind se va genera automat metoda:

```
protected void Button1_Click(object sender, EventArgs e)
{

}
```

atunci când butonul este apăsat se face o submitere a formularului către server și codul conținut în metoda de mai sus va fi executat. Codul va fi:

```
private void Button1_Click(object sender, System.EventArgs e)
{
 lblHelloWorld.Text = "Text schimbat";
}
```

La apăsarea butonului textul din componenta lblHelloWorld se va schimba, iar pagina nouă va fi trimisă către browser. Reîmprospătări succesive ale paginii (refresh) vor duce la afișarea aceluiași “Text schimbat”, deoarece viewstate-ul conține valoarea modificată de apăsarea butonului.

## 10.4 Controale server

### 10.4.1 Postback

Postback apare de câte ori se transmite pagina de la client înapoi la server. Există două modalități de realizarea a acestei trimiteri înapoi: prin folosirea unor elemente de tip `<input type="submit">` sau `<input type="image">` respectiv prin script executat pe client. De exemplu, dacă se trage pe formă un buton, codul generat de designer va fi:

```
<form method="post" runat="server" id="Form1">
 <asp:Button id="Button1" runat="server" Text="Button 1"></asp:Button>
</form>
```

iar pe browser se va genera:

```
<form name="Form1" method="post" action="MyASPNETPage.aspx" id="Form1">
 <input type="submit" name="Button1" value="Button 1" id="Button1" />
</form>
```

Dacă pentru un formular nu se specifică partea de action, aceasta implicit va fi aceeași pagină.

Trimiterea formularului prin cod client s-ar face astfel:

```
<script language="javascript">
 function SubmitTheForm()
 {
 document.forms["Form1"].submit();
 }
</script>
```

iar proprietatea *onclick* a unui contro de tip HTML se setează la valoarea *SubmitTheForm()*:

```
<INPUT type="button" value="Button" onclick="SubmitTheForm()">
```

În cazul controalelor Web de tip server a căror modificare de stare trebuie să ducă la postback (de exemplu la selectarea unui item dintr-un combobox), se setează proprietatea *AutoPostBack* pe *true*, se setează o metodă pentru evenimentul de schimbare a conținutului (în cazul nostru: *SelectedIndexChanged*) iar pagina HTML generată pe client va conține o funcție javascript de submitere a formularului:

```
<script type="text/javascript">
<!--
var theForm = document.forms['form1'];
if (!theForm) {
 theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
 if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
 theForm.__EVENTTARGET.value = eventTarget;
 theForm.__EVENTARGUMENT.value = eventArgument;
 theForm.submit();
 }
}
// -->
</script>
```

unde *\_\_EVENTTARGET* și *\_\_EVENTARGUMENT* sunt două câmpuri ascunse. Combobox-ul arată astfel în codul HTML client:

```
<select name="DropDownList1" onchange="javascript:setTimeout('__doPostBack(\'\
<option value="Opt 1">Opt 1</option>
<option value="Opt 2">Opt 2</option>
</select>
```

### 10.4.2 Data Binding

Data Binding permite legarea unei proprietăți a unui control la o sursă de date, precum un câmp în baza de date. Această legare se poate face scriind expresii în interiorul unei declarații de control în fișierul .aspx sau în code-behind.

Pentru a se lega direct în fișierul .aspx se folosește tag-ul `<%# %>` (expresie Data Binding). Această metodă se aplică pentru orice control care nu este listă precum Label, TextBox, CheckBox. Când Data Binding se face pentru un control de tip listă (DropDownList sau CheckBoxList) se va folosi proprietatea DataSource a controlului la o sursă de date oarecare (e.g. DataSet). Aceasta poate fi făcută în partea de code-behind sau într-un bloc de tip script din fișierul .aspx.

Toate controalele care au posibilități de Data Binding implementează o metodă numită DataBind(). Aceasta se ocupă de evaluarea expresiilor de binding în fișierul .aspx sau pentru sursele de date specificate în proprietatea DataSource. Este important de reținut că expresiile Data Binding și proprietățile DataSource nu sunt evaluate până când nu se apelează metoda DataBind() ale controalelor cu care sunt asociate. Deoarece Page este el însuși un control, are o implementare de metodă DataBind(), care când este apelată va duce la apelul tuturor metodelor DataBind() pentru controalele copil.

## 10.5 Controale Web

Controalele Web reprezintă o modalitate ușoară de a interacționa cu serverul. Nu există o relație univocă între controale web și elemente Html, ci controalele web vor randa pe parte de client cod Html specific browserului. Clasa Page are o proprietate nestatică numită *ClientTarget* care permite definirea browserului client. Valorile pentru proprietate sunt:

- Auto - se va face autodetectarea browserului
- DownLevel - se presupune că browserul știe doar Html 3.2 sau mai puțin
- UpLevel - browserul știe Html 4 sau mai mult, CSS, EcmaScript, Microsoft Document Object Model

Orice control Web este derivat din System.Web.UI.WebControls care este derivat din System.Web.UI.Control. Aceasta din urmă pune la dispoziție partea de Data Binding, viewstate, evenimente precum Init, Load, Unload,

etc. Clasa `WebControl` conține proprietăți de stil precum `background`, `height`, `width`.

### 10.5.1 Label

Conține text de tip `read-only`, asignat static sau prin `Data Binding`. Se generează pe browser ca fiind un element de tip *span*. Pentru crearea unei etichete cu text predefinit se trage un control `Web` de pe toolbar și se pot seta proprietăți ale lui, ceea ce ar putea genera următorul cod în designer:

```
<asp:Label id=Label1 runat="server" Width="70px" Height="50px">
 This is a label
</asp:Label>
```

Dacă se dorește folosirea `DataBinding`, atunci se poate scrie:

```
<asp:Label id=Label1 style="Z-INDEX: 101; LEFT: 10px; POSITION:
 absolute; TOP: 10px" runat="server"
 text="<%# DateTime.Now.ToString() %>">
</asp:Label>
```

Pentru ca valoarea să fie efectiv calculată și afișată pe parte de browser trebuie chemată metoda *DataBind()* pentru obiectul etichetă; acest lucru se poate face în partea de încărcare a paginii:

```
protected void Page_Load(object sender, System.EventArgs e)
{
 Label1.DataBind();
}
```

### 10.5.2 Button

Permite crearea unui buton de tip `submit` și care poate să determine executarea unei metode `C#` pe parte de client, pe baza unui event handler definit în `code-behind`.

În cazul în care se dorește crearea mai multor butoane care să fie procesate pe server, e nevoie de un mecanism care să facă diferențierea lor, astfel încât să se poată determina care a fost butonul apăsător. Acest lucru se face prin proprietățile *CommandName* și *CommandArgument* de tip `string`. Event handlerul pentru butoane poate să fie:

```
private void Button_Command(object sender,
 System.Web.UI.WebControls.CommandEventArgs e)
```

```
{
 //This Is a Command event handler
}
```

unde obiectul *e* are proprietățile *CommandName* și *CommandArgument* care vor da valorile setate pentru controale.

### 10.5.3 LinkButton

Se comportă ca și un buton; codul Html generat va fi de forma `<a>` iar la apăsare va apela funcția Javascript `__doPostBack` prezentată mai sus.

### 10.5.4 TextBox

Dă posibilitatea de a introduce text într-un formular. Se poate folosi în trei moduri: *SingleLine*, *MultiLine* sau *Password*. Prezintă proprietăți de tipul *Columns*, *Rows*, *Wrap*, *MaxLength*, *ReadOnly*, *Text*, *AutoPostBack* și un eveniment *TextChanged*. Dacă proprietatea *AutoPostBack* este setată la true, pagina va fi automat retransmisă la server dacă se iese din controlul curent și textul a fost schimbat; mecanismul se bazează pe aceeași funcție Javascript ca la *LinkButton*.

### 10.5.5 CheckBox

Determină apariția unui checkbox; proprietățile de interes sunt *Checked*, *Text*, *TextAlign*, *AutoPostBack* iar evenimentul detectat este *CheckedChanged*.

### 10.5.6 RadioButton

Derivată din *CheckBox*, are în plus proprietatea *GroupName* care permite gruparea mai multor astfel de butoane.

### 10.5.7 DropDownList

Permite crearea unei liste de opțiuni stocate sub numele de nume și valori. Toate item-urile conținute sunt accesibile via proprietatea *Items* de tip *ListItemCollection*. Fiecare element al unei astfel de colecții este de tip *ListItem* și are proprietățile *Text*, *Value* de tip String și *Selected* de tip boolean. Lista de opțiuni se poate crea fie prin introducere directă în proprietatea *Items*, fie prin Data Binding. Evenimentul care se poate detecta este *SelectedIndexChanged* și se poate procesa pe server.



# Curs 11

## ASP.NET (2)

### 11.1 Controale de validare

Controalele de validare sunt utilizate pentru a face verificari asupra îndeplinirii unor condiții pentru conținutul controalelor de introducere a datelor. Această validare se poate face pe server sau (de preferat în majoritatea cazurilor) pe client (dacă acesta suportă DHTML). Validări suplimentare mai complexe pot apărea și pe server, chiar dacă au fost efectuate validări primare pe client. Unui control de preluare a datelor i se pot asocia oricâte controale de validare. De exemplu, se poate folosi un control de tipul *RequiredFieldValidator* și altul care să valideze conținutul folosind expresii regulate; doar primul control poate determina dacă s-a introdus ceva, celelalte returnând automat valoare de adevăr dacă controlul de intrare nu conține nimic.

Orice astfel de control de validare este derivat din clasa *BaseValidator*, care prezintă proprietatea *ControlToValidate* în care se depune ca valoare ID-ul controlului pentru care se face validarea. Proprietatea *Display* determină dacă se rezervă loc pe pagină pentru a se afișa mesajul de eroare. Poate avea valorile *Static* și *Dynamic*: în primul caz se rezervă suficient spațiu pentru mesajul de eroare, chiar dacă acesta nu este vizibil; în al doilea caz dacă mesajul de eroare este arătat, atunci pot apărea deplasări ale componentelor de pe pagină pentru a face loc textului de eroare. Mesajul de eroare se definește în proprietatea *ErrorMessage*.

Proprietatea *EnableClientScript* setată pe *true* va duce la generarea de cod scripting pentru client (JavaScript sau VBScript); dacă proprietatea este setată pe *false* se va face validarea doar pe server. *Trebuie reținut faptul că validarea se face întotdeauna și pe server.* Pentru un client de tip non-DHTML, pentru un asemenea control nu se va genera nici un cod scripting client, ci atunci când formularul este trimis către server se va face testare va-

validității datelor introduse; dacă ceva nu corespunde criteriilor se va retrimite pagina către client împreună cu mesajele de eroare scrise.

În acest context spunem că clasa *Page* conține o proprietate *IsValid* care va fi setată la *true* dacă toate validatoarele sunt îndeplinite pe parte de server sau *false* în caz contrar.

Controalele de validare sunt: *RequiredFieldValidator*, *RegularExpressionValidator*, *ValidationSummary*, *RangeValidator*, *CompareValidator*, *CustomValidator*.

### 11.1.1 RequiredFieldValidator

Forțează utilizatorul să introducă ceva într-un control de introducere. Dacă se încearcă submiterea formularului iar câmpul pentru care s-a setat validatorul nu are valoare introdusă se va semnala eroare prin apariția mesajului definit de utilizator în proprietatea *ErrorMessage*. Codul JavaScript care se efectuează aceste validări este scris în fișierul sursă C:\Inetpub\wwwroot\aspnet\_client\system\_web\1\_1\_4322\WebUIValidation.js.

### 11.1.2 RegularExpressionValidator

Verifică pe baza unei expresii regulate dacă valoarea introdusă într-un câmp este corectă. Pe parte de client se va folosi sintaxă de regex-uri de tip JavaScript, pe parte de server se va folosi clasa *Regex*<sup>1</sup>. În Visual Studio sunt definite câteva expresii regulate care se pot aplica pentru diferite scopuri: adrese de email, numere de telefoane din diverse țări, URL, etc.

### 11.1.3 RangeValidator

Verifică dacă valoarea dintr-o intrare este cuprinsă între un minim și un maxim. Pentru conținut și extreme tipul poate fi *String*, *Integer*, *Double*, *Date* și *Currency*, specificat în proprietatea *Type* a controlului. Valorile minime și maxime admise se dau în proprietățile *MinimumValue* și *MaximumValue*.

### 11.1.4 CompareValidator

Se folosește pentru a compara conținutul unui control cu conținutul altui control sau cu o valoare fixă, predefinită. Dacă se dorește compararea cu

---

<sup>1</sup>Pentru o colecție bogată de expresii regulate, a se vedea și [www.regexlib.com](http://www.regexlib.com)



valoarea dintr-un alt control, se setează adecvat proprietatea *ControlToCompare*, iar dacă se face compararea cu o valoare predefinită atunci se setează proprietatea *ValueToCompare*. De asemenea se poate specifica tipul de dată pentru care se face compararea, ca la *RangeValidator*. Proprietatea *Operator* specifică operatorul folosit la comparare: *Equal* (implicit), *NotEqual*, *GreaterThan*, *GreaterThanEqual*, *LessThan*, *LessThanEqual*, și *DataTypeCheck* (operator unar, care verifică validitatea datei calendaristice specificate).

### 11.1.5 CustomValidator

Permite crearea de validatoare de către programator, care se pot executa atât pe client cât și pe server. Pentru validarea pe server trebuie dat un delegat de tipul:

```
void HandlerName (object source, ServerValidateEventArgs args)
```

Al doilea parametru are proprietatea *Value* care dă acces pe server la valoarea introdusă pe client și proprietatea *IsValid* de tip bool care trebuie setată pe *true* dacă validarea s-a făcut cu succes sau *false* în caz contrar. Această valoare afectează valoarea proprietății *IsValid* a obiectului de tip *Page*.

Motivul pentru care s-ar folosi un asemenea validator ar fi că verificarea se poate face numai pe server (de exemplu se verifica dacă numele și parola unui utilizator se regasesc în baza de date). Dacă însă se dorește validarea și pe client folosind JavaScript, atunci se va seta în mod adecvat proprietatea *ClientValidationFunction*.

Exemplu: pentru validarea pe server a valorii introduse într-un câmp, se poate scrie codul C#:

```
private void vldUserIDCstm_ServerValidate(object source,
 ServerValidateEventArgs args)
{
 if (args.Value == "JoeHealy" || args.Value == "SpikePierson" ||
 args.Value == "AndyJohnston")
 {
 args.IsValid = false;
 }
 else
 {
 args.IsValid = true;
 }
}
```

Pentru a se face aceeași verificare și pe client, se poate scrie codul următor la sfârșitul paginii aspx:

```
<script language="JavaScript">
function vldUserIDCstm_ClientValidate(source, args)
{
 if (args.Value = "JoeHealy" || args.Value = "SpikePierson" ||
 args.Value = "AndyJohnston")
 {
 args.IsValid=false;
 }
 else
 {
 args.IsValid=true;
 }
}
</script>
```

Numele funcției JavaScript este setat în proprietatea *ClientValidationFunction*.

### 11.1.6 ValidationSummary

Nu este un control de validare propriu zis, fiind derivat din clasa *WebControl*. Este folosit pentru a afișa mesajele de eroare pentru toate controalele de validare într-un singur loc. Proprietatea *DisplayMode* este folosită pentru a controla formatul în care se vor afișa mesajele de eroare, având valorile posibile: *List*, *BulletList*, *SingleParagraph*. De asemenea există proprietatea *HeaderText* care permite afișarea unui antet deasupra listei de erori.

Mesajele de eroare pot fi afișate într-unul din 2 moduri sau în ambele. Ele pot fi afișate în pagină de orice browser, fie el *UpLevel* fie *DownLevel*. În plus, un browser *UpLevel* poate să afișeze mesajul (și) într-o fereastră de eroare creată de browser. Dacă proprietatea de tip bool *ShowSummary* este setată pe true, atunci se vor afișa mesajele de eroare în pagina HTML. Dacă proprietatea de tip bool *ShowMessageBox* este pusă pe true, atunci va apărea o fereastră cu lista de erori; în acest din urmă caz, pentru un browser de tip *DownLevel* mesajul nu va fi arătat în fereastră.

## 11.2 Comunicarea cu browserul

Pentru trimiterea de date către client se poate folosi pe parte de server obiectul predefinit *Response* de tip *HttpResponse*. Deși ASP.NET pune la

dispoziție mecanisme suficient de complexe pentru a nu necesita folosirea masivă a acestui mecanism, *Response* este util pentru:

1. pagini foarte simple sau debugging;
2. modificarea mecanismului de buffering;
3. redirectarea clientului;
4. managementul mecanismului de caching

### 11.2.1 Generarea programatică conținutului

În unele cazuri răspunsul care se trimite de la server la browserul client poate fi suficient de simplu pentru a nu necesita utilizarea de controale. Utilitatea unei asemenea metode este limitată, dar există situații în care calea se dovedește a fi bună.

Exemplu:

```
protected void Page_Load(object sender, System.EventArgs e)
{
 Response.Write(string.Format("The date is: {0}
",
 DateTime.Now.ToShortDateString()));
 Response.Write(string.Format("<i>The time is: {0}</i>
",
 DateTime.Now.ToShortTimeString()));
}
```

De asemenea se permite trimiterea conținutului unor fișiere către browser, via metoda *WriteFile*.

### 11.2.2 Redirectarea

Se folosește pentru cazul în care pagina care se trimite clientului nu este cea curentă, ci o alta. De exemplu, pentru cazul în care se dorește redirectarea utilizatorului către o pagină de login. Redirectarea trebuie făcută înainte ca vreun header să fie trimis către client.

Exemplu:

```
Response.Redirect("http://mysite.com/myBusiness");
```

Această redirectare se face trimitând clientului un cod în care i se indică noua pagină pe care trebuie să o ceară de la server.

## 11.3 Cookies

Cookie-urile reprezintă informație stocată pe client. Deși nu reprezintă un mecanism garantat pentru lucrul cu clientul, în unele cazuri pot fi suficient de utile pentru o aplicație. Există 4 clase de lucru cu cookie-urile:

- *HttpCookie*
- *HttpCookieCollection*
- *HttpResponse*
- *HttpRequest*

Obiectul *Response* are o colecție numită *Cookies*. Când un browser face o cerere către un server, va trimite prin intermediul headerelor toată colecția de cookie-uri care aparțin acelui site. Această colecție se încarcă în *Cookies*.

Atunci când se trimite un cookie de pe server pe client se poate preciza dacă acesta va fi valabil doar pe perioada cât este deschis browserul (așa-numitele cookie-uri *de sesiune*) sau și după ce browserul este închis (cookie-uri *persistente* - rezistă până la o dată a expirării setată sau pentru totdeauna). Primul tip de cookie-uri sunt utile în medii în care nu există intimitate a utilizatorului, un cont fiind folosit de către mai mulți; al doilea tip se poate folosi pentru stocarea informațiilor (precum preferințele) de la o vizită la alta.

Diferența dintre cele două din punct de vedere programatic constă în absența sau prezența unui timp de expirare. Cele de tip sesiune nu au setat nici un timp de expirare (și se rețin în memoria RAM a calculatorului), pe când celelalte pot avea un timp setat în viitor (fiind persistate pe disc).

Exemplu de creare a unui cookie de tip sesiune:

```
protected void Page_Load(object sender, System.EventArgs e)
{
 HttpCookie tempcookie = new HttpCookie("myCookie");
 tempcookie.Values.Add("userid", "1250");
 Response.Cookies.Add(tempcookie);
}
```

Pentru crearea unui cookie persistent, se poate proceda ca mai jos:

```
protected void Page_Load(object sender, System.EventArgs e)
{
 HttpCookie perscookie = new HttpCookie("myCookie");
 perscookie.Expires = Convert.ToDateTime("24/05/05 16:00");
}
```

```
perscookie.Values.Add("userid", "1250");
Response.Cookies.Add(perscookie);
}
```

Accesarea acestui cookie se face pe server prin intermediul obiectului *Request*:

```
HttpCookie cookie = Request.Cookies["myCookie"];
if (cookie != null)
{
 string s = cookie.Values["userid"].ToString();
 ...
}
```

## 11.4 Fișierul de configurare al aplicației Web

Diverse setări ale aplicației pot fi menținute într-un fișier de tip XML numit *Web.config*. Structura și modul de accesare a conținutului acestuia este asemănătoare cu cea prezentată în secțiunea 7.4.4. Deosebit este însă faptul pentru o aplicație Web această configurare respectă o anumită ierarhie. La rădăcina acestei ierarhii se află fișierul *machine.config* aflat în directorul %windir%\Microsoft.NET\Framework\Version\CONFIG. Setările de aici pot fi suprascrise de setări aflate în directoarele *Web.config* unei aplicații ASP.NET.

## 11.5 Fișierul *global.asax*

Este un fișier opțional care prezintă setări relative la aplicația ASP.NET (mai exact, pentru obiectul reprezentând aplicația). Fișierul se găsește în rădăcina aplicației, este compilat ca o clasă care este derivată din *HttpApplication*, fiind recompilată automat la fiecare modificare a sa.

Metodele care se pot implementa în această clasă sunt:

- *Application\_Start* - apelată atunci când un client cere pentru prima oară o pagină de la această aplicație; folosită pentru inițializarea unor resurse (grupuri de conexiuni, valori din regiștri, etc care sunt accesate de către părți din aplicație).
- *Session\_Start* - ori de câte ori se pornește o sesiune
- *Application\_BeginRequest* - apelat ori de câte ori un client cere o pagină oarecare de la server; s-ar putea folosi la contorizarea numărului de accese la pagină

- *Application\_EndRequest* - apelat la sfârșitul oricărei cereri către server
- *Application\_AuthenticateRequest* - apelată la fiecare cerere de autentificare a unui utilizator
- *Application\_Error* - apelată ori de câte ori apare o eroare care nu a fost procesată
- *Session\_End* - la terminarea unei sesiuni prin expirare
- *Application\_End* - apare la o resetare a serverului sau atunci când se recompilează aplicația ASP.NET.

## 11.6 Managementul sesiunii

Sesiunea este un mecanism prin care se creează impresia unei conexiuni permanente de la client la server. O sesiune este stocată pe server și va reține toate obiectele necesare bunei funcționări pe perioada respectivă (de exemplu poate conține coșul de cumpărături pentru un client, ID-ul celui client, etc). Ori de câte ori un client cere prima pagină a unei aplicații Web, se creează un obiect de sesiune pe server identificat printr-un număr unic care este de asemenea trimis clientului printr-un cookie. Această sesiune este atașată doar browserului care a cerut crearea ei; dacă acesta se închide, atunci vechea sesiune este inaccesibilă. De fiecare dată când se cere o pagină de la server (după ce sesiunea a fost creată), se cere id-ul stocat în cookie-ul de pe client și se află care este sesiunea atașată.

Stocarea unei valori în sesiune trebuie să se facă folosind o cheie (unică pe acea sesiune) și o valoare atașată (de ori ce tip). Exemplu:

```
Session["username"] = "jsmiley";
```

Dacă cheia *username* nu există, se va crea și i se va atașa string-ul numit; dacă există, valoarea sa se va suprascrie. Recuperarea valorii anterioare se face cu:

```
String uname = (string)Session["username"];
```

Menționăm că în sesiune se pot stoca orice obiecte, nu doar șiruri de caractere. Dacă serverul web se repornește, toate valorile din acea sesiune se pierd; există alternativa de a stoca sesiunile pe un alt server sau chiar pe un server SQL. Ștergerea unei chei și a valori atașate din sesiune se face folosind metodele *Session.Remove(String key)*, *Session.RemoveAt(int index)*. Golirea sesiunii se face cu *Session.RemoveAll()* sau *Session.Clear()*.

## 11.7 ViewState

*ViewState* reprezintă o colecție de valori care se stochează pe client, dar nu prin intermediul cookie-urilor, ci pe baza unui câmp de tip hidden din pagina HTML trimisă clientului. Este folosită în special pentru a reține valorile controalelor Web de-a lungul mai multor post-back-uri.

Spre deosebire de obiectul *Session* care poate stoca orice tip de obiect, obiectul *ViewState* (de tip *StateBag*) poate să conțină doar string-uri.

Exemplu:

```
ViewState["uname"] = "Joe";
..
String surname = (string)ViewState["uname"];
```

Se poate șterge selectiv din obiectul *ViewState* prin metoda *Remove(String cheie)*.

## 11.8 Application

Obiectul *Application* este foarte asemănător cu *Session*, cu deosebirea că nu este unic pentru fiecare sesiune utilizator, ci pentru întreaga aplicație. Este utilizat pentru menținerea unor obiecte cu caracter global, utile pentru orice sesiune (valori din regiștri sau valori imuabile care nu trebuie citite de fiecare dată). Stocarea de valori se face prin:

```
Application["x"] = new MyObject();
```

iar accesarea lui prin:

```
MyObject x = Application["x"];
```

## 11.9 Alte puncte de interes în aplicații ASP.NET

- crearea controalelor utilizator - a paginilor care sunt convertite în controale pentru a permite reutilizarea lor mai simplă - ex: header, footer;
- separarea codului de prezentare - pentru a permite separarea muncii între web designer și programator; separarea pe mai multe nivele ușurează menținerea codului și schimbarea lui; este sugerată folosirea a cel puțin 3 nivele: interfața utilizator, nivelul de business logic, nivelul de acces la date;

- crearea de aplicații web pentru dispozitive mobile; deși la ora actuală piața de dispozitive mobile este relativ redusă, există totuși un segment bine definit care cere astfel de aplicații. Design-ul specific și capabilitățile unui astfel de browser pot ridica probleme noi;
- politici de caching - pentru optimizarea accesului la resurse;
- autentificarea - diverse forme de securitate: formulare, .NET Passport;
- folosirea protocoalelor de comunicație securizate (SSL)
- folosirea serviciilor Web - servicii care se pot accesa în Internet; bazate pe XML și protocoale de comunicație larg răspândite, un serviciu Web este independent de platformă.



# Bibliografie

- [1] *Programming C#*, O'Reilly, Jesse Liberty, 4th edition, 2005
- [2] *Microsoft C# 2005 Step by Step*, John Sharp, Microsoft Press, 2005
- [3] *Core C# and .NET*, Stephen C. Perry, Prentice Hall, 2005
- [4] *Pro ASP.NET in C# 2005*, Mathew MacDonald and Mario Szpuszta, Apress, 2005
- [5] *C# Language Specification*, ECMA TC39/TG2, Octombrie 2002
- [6] *Professional ADO.NET Programming*, Wrox, 2001