

# Medii vizuale de programare

Lector Lucian Sasu, Ph.D.

May 30, 2012

# Cuprins

<b>1</b>	<b>Platforma Microsoft .NET</b>	<b>9</b>
1.1	Prezentare generală . . . . .	9
1.2	Arhitectura platformei Microsoft .NET . . . . .	11
1.3	Componente ale lui .NET Framework . . . . .	12
1.3.1	Common 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
<b>2</b>	<b>Tipuri predefinite, tablouri, string-uri</b>	<b>20</b>
2.1	Vedere generală asupra limbajului C# . . . . .	20
2.2	Tipuri de date . . . . .	22
2.2.1	Tipuri predefinite . . . . .	22
2.2.2	Tipuri valoare . . . . .	24
2.2.3	Tipul enumerare . . . . .	29
2.3	Tablouri . . . . .	35
2.3.1	Tablouri unidimensionale . . . . .	35
2.3.2	Tablouri multidimensionale . . . . .	37
2.4	Șiruri de caractere . . . . .	40
2.4.1	Expresii regulate . . . . .	42
<b>3</b>	<b>Clase, instrucțiuni, spații de nume</b>	<b>44</b>
3.1	Clase – vedere generală . . . . .	44
3.2	Transmiterea de parametri . . . . .	48
3.3	Conversii . . . . .	57
3.3.1	Conversii implicite . . . . .	57

3.3.2	Conversiile implicite ale expresiilor constante . . . . .	59
3.3.3	Conversii explicite . . . . .	60
3.3.4	Boxing și unboxing . . . . .	63
3.4	Declarații de variabile și constante . . . . .	64
3.5	Instrucțiuni C# . . . . .	65
3.5.1	Declarații de etichete . . . . .	65
3.5.2	Instrucțiuni de selecție . . . . .	66
3.5.3	Instrucțiuni de ciclare . . . . .	67
3.5.4	Instrucțiuni de salt . . . . .	69
3.5.5	Instrucțiunile try, throw, catch, finally . . . . .	69
3.5.6	Instrucțiunile checked și unchecked . . . . .	70
3.5.7	Instrucțiunea lock . . . . .	70
3.5.8	Instrucțiunea using . . . . .	70
3.6	Spații de nume . . . . .	71
3.6.1	Declarații de spații de nume . . . . .	72
3.6.2	Directiva using . . . . .	73
3.7	Declararea unei clase . . . . .	77
3.8	Membrii unei clase . . . . .	78
3.9	Constructorii de instanță . . . . .	79
3.10	Câmpuri . . . . .	79
3.10.1	Câmpuri instanțe . . . . .	80
3.10.2	Câmpuri statice . . . . .	80
3.10.3	Câmpuri readonly . . . . .	81
3.10.4	Câmpuri volatile . . . . .	81
3.10.5	Inițializarea câmpurilor . . . . .	82
3.11	Constante . . . . .	82
3.12	Metode . . . . .	83
3.12.1	Metode statice și nestatice . . . . .	84
3.12.2	Metode externe . . . . .	84
<b>4</b>	<b>Clase (continuare)</b>	<b>85</b>
4.1	Proprietăți . . . . .	85
4.2	Indexatori . . . . .	91
4.3	Operatori . . . . .	97
4.3.1	Operatori unari . . . . .	97
4.3.2	Operatori binari . . . . .	99
4.3.3	Operatori de conversie . . . . .	100
4.3.4	Exemplu: clasa Fraction . . . . .	101
4.4	Constructor static . . . . .	103
4.5	Clase imbricate . . . . .	104
4.6	Destructorii . . . . .	107

4.7	Clase statice . . . . .	110
4.8	Specializarea și generalizarea . . . . .	111
4.8.1	Specificarea moștenirii . . . . .	112
4.8.2	Apelul constructorilor din clasa de bază . . . . .	113
4.8.3	Operatorii <i>is</i> și <i>as</i> . . . . .	114
4.9	Clase <i>sealed</i> . . . . .	115
<b>5</b>	<b>Clase, structuri, interfețe, delegați</b>	<b>116</b>
5.1	Polimorfismul . . . . .	116
5.1.1	Polimorfismul parametric . . . . .	116
5.1.2	Polimorfismul ad-hoc . . . . .	116
5.1.3	Polimorfismul de moștenire . . . . .	117
5.1.4	<i>Virtual</i> și <i>override</i> . . . . .	118
5.1.5	Modificatorul <i>new</i> pentru metode . . . . .	119
5.1.6	Metode <i>sealed</i> . . . . .	122
5.1.7	Exemplu folosind <i>virtual</i> , <i>new</i> , <i>override</i> , <i>sealed</i> . . . . .	123
5.2	Clase și metode abstracte . . . . .	125
5.3	Tipuri parțiale . . . . .	126
5.4	Structuri . . . . .	128
5.4.1	Structuri sau clase? . . . . .	132
5.5	Interfețe . . . . .	132
5.5.1	Clase abstracte sau interfețe? . . . . .	138
5.6	Tipul delegat . . . . .	139
5.6.1	Utilizarea delegaților pentru a specifica metode la runtime	140
5.6.2	Delegați statici . . . . .	143
<b>6</b>	<b>Metode anonime. Evenimente. Excepții.</b>	<b>144</b>
6.1	Metode anonime . . . . .	144
6.1.1	Multicasting . . . . .	146
6.2	Evenimente . . . . .	149
6.2.1	Publicarea și subscrierea . . . . .	150
6.2.2	Evenimente și delegați . . . . .	150
6.2.3	Comentarii . . . . .	156
6.3	Tratarea excepțiilor . . . . .	157
6.3.1	Tipul <i>Exception</i> . . . . .	157
6.3.2	Aruncarea și prinderea excepțiilor . . . . .	158
6.3.3	Reîncercarea codului . . . . .	169
6.3.4	Compararea tehnicilor de manipulare a erorilor . . . . .	171
6.3.5	Sugestie pentru lucrul cu excepțiile . . . . .	172

<b>7</b>	<b>Colecții și clase generice</b>	<b>173</b>
7.1	Colecții	173
7.1.1	Iteratori pentru colecții	176
7.1.2	Colecții de tip listă	177
7.1.3	Colecții de tip dicționar	178
7.2	Crearea unei colecții	179
7.2.1	Colecție iterabilă (stil vechi)	179
7.2.2	Colecție iterabilă (stil nou)	182
7.3	Clase generice	188
7.3.1	Metode generice	188
7.3.2	Tipuri generice	189
7.3.3	Constrângeri asupra parametrilor de genericitate	190
7.3.4	Interfețe și delegați generici	191
7.4	Colecții generice	192
7.4.1	Probleme cu colecțiile de obiecte	192
7.4.2	Colecții generice	193
7.4.3	Metode utile în colecții	193
<b>8</b>	<b>ADO.NET</b>	<b>197</b>
8.1	Ce reprezintă ADO.NET?	197
8.2	Furnizori de date în ADO.NET	198
8.3	Componentele unui furnizor de date	199
8.3.1	Clasele <i>Connection</i>	199
8.3.2	Clasele <i>Command</i>	200
8.3.3	Clasele <i>DataReader</i>	200
8.3.4	Clasele <i>DataAdapter</i>	200
8.3.5	Clasa <i>DataSet</i>	200
8.4	Obiecte <i>Connection</i>	200
8.4.1	Proprietăți	201
8.4.2	Metode	203
8.4.3	Evenimente	203
8.4.4	Stocarea stringului de conexiune în fișier de configurare	203
8.4.5	Gruparea conexiunilor	205
8.4.6	Mod de lucru cu conexiunile	205
8.5	Obiecte <i>Command</i>	206
8.5.1	Proprietăți	207
8.5.2	Metode	208
8.5.3	Utilizarea unei comenzi cu o procedură stocată	210
8.5.4	Folosirea comenzilor parametrizate	211
8.6	Obiecte <i>DataReader</i>	213
8.6.1	Proprietăți	213

8.6.2	Metode . . . . .	214
8.6.3	Crearea și utilizarea unui <i>DataReader</i> . . . . .	214
8.6.4	Utilizarea de seturi de date multiple . . . . .	216
8.6.5	Seturi de date cu tip . . . . .	216
<b>9</b>	<b>ADO.NET (2)</b>	<b>218</b>
9.1	Obiecte <i>DataAdapter</i> . . . . .	218
9.1.1	Metode . . . . .	219
9.1.2	Proprietăți . . . . .	220
9.2	Clasa <i>DataSet</i> . . . . .	221
9.2.1	Conținut . . . . .	221
9.2.2	Clasa <i>DataTable</i> . . . . .	222
9.2.3	Relații între tabele . . . . .	224
9.2.4	Popularea unui <i>DataSet</i> . . . . .	224
9.2.5	Clasa <i>DataTableReader</i> . . . . .	225
9.2.6	Propagarea modificărilor către baza de date . . . . .	226
9.3	Tranzacții în ADO.NET . . . . .	228
9.4	Lucrul generic cu furnizori de date . . . . .	230
9.5	Tipuri nulaibile . . . . .	233
<b>10</b>	<b>LINQ (I)</b>	<b>235</b>
10.1	Elemente specifice C# 3.0 . . . . .	235
10.1.1	Proprietăți implementate automat . . . . .	235
10.1.2	Inițializatori de obiecte . . . . .	236
10.1.3	Inițializatori de colecții . . . . .	238
10.1.4	Inferența tipului . . . . .	239
10.1.5	Tipuri anonime . . . . .	240
10.1.6	Metode parțiale . . . . .	241
10.1.7	Metode de extensie . . . . .	242
10.2	Generalități . . . . .	244
10.3	Motivație . . . . .	245
10.3.1	Codul clasic ADO.NET . . . . .	246
10.3.2	Nepotrivirea de paradigme . . . . .	247
10.4	LINQ to Objects: exemplificare . . . . .	248
10.4.1	Expresii lambda . . . . .	249
10.5	Operatori LINQ . . . . .	250
<b>11</b>	<b>LINQ (II)</b>	<b>253</b>
11.1	LINQ to Objects . . . . .	253
11.1.1	Filtrarea cu <i>Where</i> . . . . .	255
11.1.2	Operatorul de proiecție . . . . .	256

11.1.3	Operatorul <b>let</b>	257
11.1.4	Schimbarea ordinii	257
11.1.5	Partiționare	259
11.1.6	Concatenarea	259
11.1.7	Referirea de elemente din secvențe	259
11.1.8	Operatorul <b>SelectMany</b>	260
11.1.9	Joncțiuni	263
11.1.10	Grupare	263
11.1.11	Agregare	264
11.2	Exerciții	266
<b>12</b>	<b>Atribute. Fire de execuție</b>	<b>270</b>
12.1	Atribute	270
12.1.1	Generalități	270
12.1.2	Atribute predefinite	272
12.1.3	Exemplificarea altor atribute predefinite	274
12.1.4	Atribute definite de utilizator	277
12.2	Fire de execuție	280
12.3	Managementul thread-urilor	280
12.3.1	Pornirea thread-urilor	280
12.3.2	Metoda <i>Join()</i>	283
12.3.3	Suspendarea firelor de execuție	283
12.3.4	Omorârea thread-urilor	284
12.3.5	Sugerarea priorităților firelor de execuție	288
12.3.6	Fire în fundal și fire în prim-plan	288
12.4	Sincronizarea	289
12.4.1	Clasa <i>Interlocked</i>	292
12.4.2	Instrucțiunea <i>lock</i>	293
12.4.3	Clasa <i>Monitor</i>	294
<b>13</b>	<b>Noutăți în C# 4.0</b>	<b>297</b>
13.1	Parallel LINQ	297
13.2	Parametri cu nume și parametri opționali	299
13.3	Tipuri de date dinamice	301
13.4	ExpandableObject	303
13.5	COM Interop	304
13.6	Covarianță și contravarianță	306
13.7	Clasele <b>BigInteger</b> și <b>Complex</b>	307
13.8	Clasa <b>Tuple</b>	308

<b>14 Fluxuri</b>	<b>309</b>
14.1 Sistemul de fişiere	309
14.1.1 Lucrul cu directoarele: clasele <i>Directory</i> şi <i>DirectoryInfo</i>	310
14.1.2 Lucrul cu fişierele: clasele <i>FileInfo</i> şi <i>File</i>	312
14.2 Citirea şi scrierea datelor	317
14.2.1 Clasa <i>Stream</i>	317
14.2.2 Clasa <i>FileStream</i>	319
14.2.3 Clasa <i>MemoryStream</i>	319
14.2.4 Clasa <i>BufferedStream</i>	320
14.2.5 Clasele <i>BinaryReader</i> şi <i>BinaryWriter</i>	321
14.2.6 Clasele <i>TextReader</i> , <i>TextWriter</i> şi descendentele lor	322
14.3 Operare sincronă şi asincronă	324
14.4 Stream-uri Web	327
14.5 Serializarea	327
14.5.1 Crearea unui obiect serializabil	327
14.5.2 Serializarea	328
14.5.3 Deserializarea unui obiect	328
14.5.4 Date tranziente	329
14.5.5 Operaţii la deserializare	329
Bibliografie	331



# Curs 1

## Platforma Microsoft .NET

### 1.1 Prezentare generală

Platforma .NET 4.0 este un cadru de dezvoltare a softului pentru sisteme de operare de tip Windows. Include o bibliotecă de mari dimensiuni, sprijină câteva limbaje ce pot interopera<sup>1</sup>. Platforma permite realizarea, distribuirea și rularea aplicațiilor de tip forme Windows, aplicații WEB și servicii WEB. Platforma constă în trei părți principale: Common Language Runtime, clasele specifice platformei și ASP.NET. O infrastructură ajutătoare, .NET Compact Framework este un set de interfețe de programare care permite dezvoltatorilor realizarea de aplicații pentru dispozitive mobile precum telefoane inteligente și PDA-uri.

.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 în câteva grupe de produse:

1. *Unelte de dezvoltare* - un set de limbaje (C#, Visual Basic .NET, J#<sup>2</sup>, C++/CLI, JScript.NET, Objective-C, IronPython, Smalltalk, Eiffel, Perl, Fortran, Cobol, Lisp, Haskell, Pascal, RPG etc.), un set de medii de dezvoltare (Visual Studio .NET, Visio), infrastructura .NET Framework și o bibliotecă cuprinzătoare de clase pentru crearea serviciilor Web (via Windows Communication Framework)<sup>3</sup>, aplicațiilor Web (ASP.NET

---

<sup>1</sup>Definiția [Wikipedia](#).

<sup>2</sup>La ora actuală, suportul de J# este restras din Visual Studio. Apelul de cod J# în .NET 4.0 eșuează, dacă nu sunt luate precauții speciale.

<sup>3</sup>Serviciilor Web - aplicații care oferă servicii folosind Web-ul ca modalitate de acces.

MVC, Web Forms) și a aplicațiilor Windows (Windows Forms, Windows Presentation Foundation).

2. *Servere specializate* - un set de servere Enterprise .NET: SQL Server 2008, Exchange 2007, BizTalk Server 2006, SharePoint etc., care pun la dispoziție funcționalități diverse pentru stocarea bazelor de date, email, aplicații B2B<sup>4</sup>.
3. *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 din trecutul recent cereau de multe ori o mare afinitate față de producător și prezentau 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, REST).
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 (pagini Web dinamice, dezvoltare de controale utilizator, pattern-uri de design e.g. MVC etc.).
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.

---

<sup>4</sup>Bussiness 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

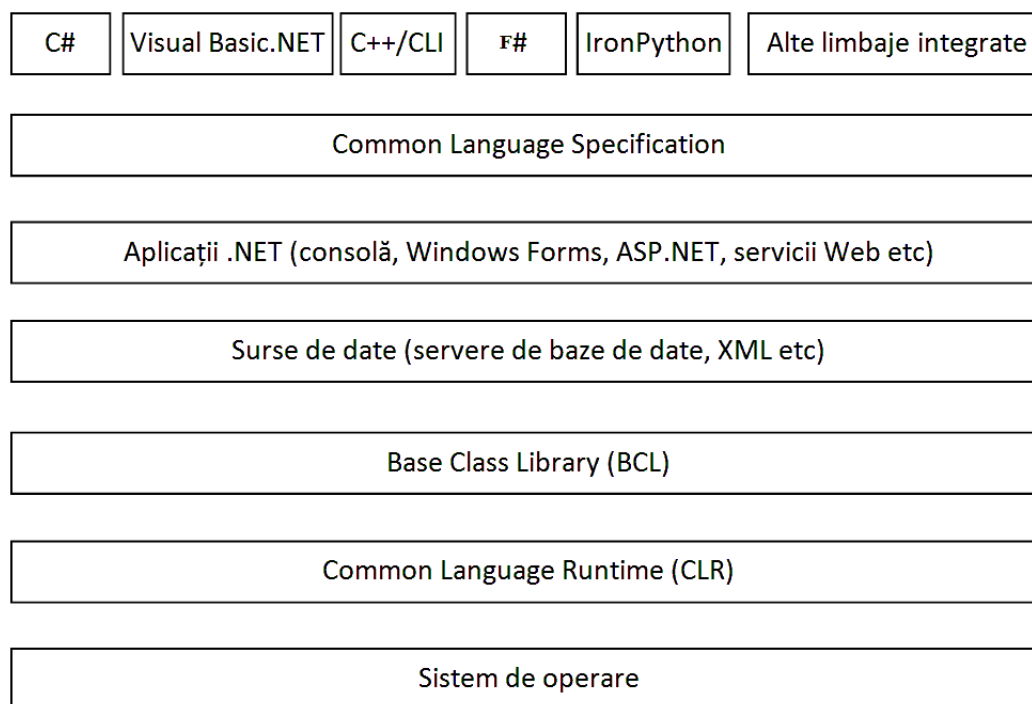


Figura 1.1: Arhitectura .NET

Figura 1.1 schematizează arhitectura platformei Microsoft .NET. Orice program scris într-unul din limbajele .NET este compilat în **Common Intermediate Language**<sup>5</sup>, 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 *multi-tier*. **Base Class Library** (BCL) există pentru a asigura funcționalitate de nivel scăzut, precum operații de intrare/ieșire, fire de execuție, lucrul cu șiruri de caractere, comunicație prin rețea etc. Aceste clase sunt reunite sub numele de **.NET Framework Class Library**. La baza tuturor se află cea mai importantă componentă a lui .NET Framework - **Common Language Runtime**, care răspunde de execuția fiecărui program. Evident, nivelul inferior este rezervat sistemului

<sup>5</sup>Anterior numit și Microsoft Intermediate Language (MSIL) sau Intermediate Language (IL).

de operare. Trebuie spus că platforma .NET nu este exclusiv dezvoltată pentru sistemul de operare Microsoft Windows, ci și pentru variante de Unix (FreeBSD sau Linux - a se vedea proiectul Mono<sup>6</sup>).

## 1.3 Componente ale lui .NET Framework

### 1.3.1 Common Intermediate Language

Una din uneltele de care dispune ingineria software este *abstractizarea*. Deseori vrem să ferim utilizatorul de anumite detalii, să punem la dispoziția altora mecanisme sau cunoștințe generale, care să permită atingerea scopului, fără a fi necesare cunoașterea tuturor dedesubturilor. 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. Bytecode-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: Common Intermediate Language. Asemănător cu bytecode-ul, CIL are trăsături OO, precum abstractizarea datelor, moștenirea, polimorfismul, sau concepte care s-au dovedit a fi necesare, precum excepțiile sau evenimentele. De remarcă 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 sunt identice: de exemplu, unele sunt case sensitive, altele nu; toate

---

<sup>6</sup>[www.go-mono.com](http://www.go-mono.com)

limbajele trebuie să aibă de acord ca la apelarea unui constructor, cel al clasei de bază trebuie să fie executat mai înainte; care sunt entitățile care pot fi suprîncărcate, faptul că tablourile trebuie să aibă indicii începînd de la 0 etc. Pentru a se asigura 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), conținînd specificații de reguli necesare pentru integrarea limbajelor.

CLS definește un set de reguli pentru compilatoarele .NET, asigurînd 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++/CLI.

### 1.3.3 Common Language Runtime

CLR este de departe cea mai importantă componentă a lui .NET Framework. Este responsabilă cu managementul și execuția codului scris în limbaje .NET, aflat în format CIL; 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 poate rezulta 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 în momentul în care este apelată pentru prima oară este tradusă în cod mașină. Codul translatat este depus într-un cache, evitînd-se astfel recompilarea ulterioară. Există 3 tipuri de compilatoare JIT:

1. *Normal JIT* - descris 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 CIL instrucțiune cu instrucțiune, fără a stoca într-un cache instrucțiunile în cod mașină.

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 aibă 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 enumerate și tratate în cadrul prezentării limbajului C#):

1. *Tipuri valoare* - în general, CLR-ul (care se ocupă de managementul și execuția codului CIL, 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 sunt tipurile primitive, structurile și enumerările. Datorită faptului că de regulă au dimensiuni mici și sunt alocate pe stivă, se manipulează eficient, eliminând costul suplimentar 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 valoare 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 imbricate 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

Metadatale înseamnă “date despre date”. În cazul .NET, ele reprezintă detalii destinate a fi citite și folosite de către platformă. Sunt stocate împreună cu codul pe care îl descriu. Pe baza metadatelor, CLR știe cum să instanțieze obiectele, cum să le apeleze metodele, cum să acceseze proprietățile. Printr-un mecanism numit reflectare, o aplicație (nu neapărat CLR) poate să interogheze aceste metadata și să afle ce expune un tip oarecare (clasă, structură etc.).

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).

### 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, numărul de 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 funcționare. 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 coresponsător. 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%\Assembly. 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 timp în programare. 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 sugera 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

în continuă creștere<sup>7</sup>. Această dezvoltare are în vedere și debugging-ul (depanarea) aplicațiilor dezvoltate în mai multe limbaje.

- **Independența de procesor și de platformă:** CIL 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 dealocare de memorie sunt în mare parte rezolvate; overhead-ul indus de către mecanismul de garbage collection este suportabil, fiind implementat în sisteme mult mai timpurii. Menționăm totuși că la ora actuală există încă posibilitatea de a avea memory leak într-o aplicație gestionată de către platforma .NET<sup>8</sup>.
- **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 folosesc XML și SOAP. Deoarece SOAP este un protocol simplu, bazat pe XML, ce folosește ca protocol de transmisie HTTP<sup>9</sup>, 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.

---

<sup>7</sup>O listă de limbaje se găsește [aici](#).

<sup>8</sup>[How to Detect and Avoid Memory and Resource Leaks in .NET Applications](#)

<sup>9</sup>Folosit la transmiterea paginilor Web pe Internet

- **Arhitectură distribuită:** Noua filosofie este de a asigura accesul la servicii Web distribuite; acestea conlucrează la obținerea informației 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#<sup>1</sup> este un limbaj de programare imperativ, obiect-orientat. Este sintactic asemănător cu Java și C++, motiv pentru care curba de învățare este 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 nume<sup>2</sup> *System*; un spațiu de nume este o colecție de tipuri sau de alte spații de nume care pot fi folosite într-un program. 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

---

<sup>1</sup>“#” se pronunță “sharp”

<sup>2</sup>Eng: namespace

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 procesului.

Pentru operațiile de intrare-ieșire cu consola se folosește clasa *Console*; pentru ea 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 precedent 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 semnaliza dacă procesul s-a încheiat cu succes sau nu<sup>3</sup>. Menționăm faptul că limbajul este case-sensitive<sup>4</sup>.

Ca metode de notare Microsoft recomandă folosirea următoarelor două convenții:

- convenție Pascal, în care prima literă a fiecărui cuvânt se scrie ca literă mare; exemplu: `LoadData`, `SaveLogFile`

---

<sup>3</sup>De exemplu în fișiere de comenzi prin testarea variabilei de mediu `ERRORLEVEL`

<sup>4</sup>Face distincție între litere mari și mici

- convenția tip “cămilă” este la fel ca precedenta, dar primul caracter al primului cuvânt nu este scris ca literă mare; exemplu: `userIdentifier`, `firstName`.

În general, convenția tip Pascal este folosită pentru tot ce este vizibil (public), precum nume de clase, metode, proprietăți etc. Parametrii metodelor și numele câmpurilor 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.

## 2.2 Tipuri de date

C# prezintă două grupuri de tipuri de date: *tipuri valoare* și *tipuri referință*. Tipurile valoare includ tipurile simple (ex. `char`, `int`, `float`)<sup>5</sup>, tipurile enumerare și structură și au ca principale caracteristici faptul că ele conțin direct datele referite și sunt alocate pe stivă sau *inline* într-o structură. Tipurile referință includ tipurile clasă, interfață, delegat și tablou, toate având proprietatea că variabilele de acest tip stochează referințe către obiectele conținute. 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 referirea vreunui spațiu de nume via directiva `using` sau calificare completă: `string`, `object`, tipurile întregi cu semn și fără semn, tipuri numerice în virgulă mobilă, `bool` și `decimal`.

Tipul `string` este folosit pentru manipularea șirurilor de caractere codificate Unicode; conținutul obiectelor de tip `string` nu se poate modifica<sup>6</sup>. 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.

---

<sup>5</sup>De fapt acestea sunt structuri, prezentate în alt curs

<sup>6</sup>Spunem despre un `string` că este *invariabil* - engl. *immutable*

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

Tabelul 2.1: Tipuri predefinite.

Tip	Descriere	Exemplu
object	rădăcina oricărui tip	object a = null;
string	o secvență de caractere Unicode	string s = "hello";
sbyte	tip întreg cu semn, pe 8 biți	sbyte val = 12;
short	tip întreg cu semn, pe 16 biți	short val = 12;
int	tip întreg cu semn, pe 32 biți	int val = 12;
long	tip întreg cu semn, pe 64 biți	long val1 = 12; long val2=34L;
byte	tip întreg fără semn, pe 8 biți	byte val = 12;
ushort	tip întreg fără semn, pe 16 biți	ushort val = 12;
uint	tip întreg fără semn, pe 32 biți	uint val = 12;
ulong	tip întreg fără semn, pe 64 de biți	ulong val1=12; ulong val2=34U; ulong val3=56L; ulong val4=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 numeric<sup>7</sup>, 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.

---

<sup>7</sup>Engl: integral type



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

## 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 2.2: Tipuri simple și corespondențele lor cu tipurile din spațiul de nume `System`.

*Tabelul 2.2*

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

Deoarece un tip simplu este un alias pentru un tip structură, 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.Parse("3.14");
```

### Tipuri întregi

C# suportă nouă tipuri întregi: sbyte, byte, short, ushort, int, uint, long, ulong și char. Acestea au următoarele dimensiuni și domenii 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 de 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*: 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 IEEE 754, care permit reprezentarea valorilor de “0 pozitiv” și “0 negativ” (sunt identice, 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  $1.5 \times 10^{-45}$  și  $3.4 \times 10^{38}$  (și din domeniul negativ corespunzător), cu o precizie de 7 cifre. Double poate reprezenta valori cuprinse între  $5.0 \times 10^{-324}$  și  $1.7 \times 10^{308}$  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<sub>optional</sub> sufix-de-tip-real<sub>optional</sub>  
 . cifre-zecimale exponent<sub>optional</sub> sufix-de-tip-real<sub>optional</sub>  
 cifre-zecimale exponent sufix-de-tip-real<sub>optional</sub>  
 cifre-zecimale sufix-de-tip-real,

unde

exponent::

e semn<sub>optional</sub> cifre-zecimale

E semn<sub>optional</sub> 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  $1.0 \times 10^{-28}$  și  $7.9 \times 10^{28}$ , 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.

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 reprezentare<sup>8</sup>, 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 indicator pe

---

<sup>8</sup>Engl: underlying type

biți, flag) al cărui număr de stări diferă de numărul de biți alocați tipului `int`:

```
enum ActionAttributes : ulong
{
    Read = 1,
    Write = 2,
    Delete = 4,
    Query = 8,
    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 fiecărui câmp poate fi specificat 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 conversie explicită:

```
...
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 minus 1, pentru un tablou unidimensional; în cadrul unui tablou multidimensional valoarea indicelui maxim este numărul de elemente de pe dimensiunea respectivă minus 1; orice depășire a indicilor duce la apariția unei excepții: `System.IndexOutOfRangeException`. O variabilă de tip tablou fie are valoare de null, fie conține adresa de memorie a unei instanțe valide.

### 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 jos<sup>9</sup>:

```
int[] sir;
```

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

---

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

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

Exemplu:

```
using System;
class Unidimensional
{
    public static int Main()
    {
        int[] sir;
        int n;
        Console.Write("Dimensiunea vectorului: ");
        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 proprietatea<sup>10</sup> `Length`, care returnează numărul *tuturor* elementelor vectorului (lucru mai 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};
```

---

<sup>10</sup>Pentru noțiunea de proprietate, vezi secțiunea 4.1, pagina 85.

### 2.3.2 Tablouri multidimensionale

C# cunoaște două tipuri de tablouri multidimensionale: rectangulare și neregulate<sup>11</sup>. Numele lor vine de la forma pe care o pot avea.

#### Tablouri rectangulare

Tablourile rectangulare au proprietatea că numărul de elemente pentru o anumită dimensiune este păstrat constant; altfel spus, acest tip de tablouri au chiar formă dreptunghiulară.

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

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

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

rezultând un tablou cu 2 linii și 3 coloane; fiecare linie are exact 3 elemente și acest lucru nu se poate schimba pentru tabloul declarat. 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;
            }
        }
    }
}
```

---

<sup>11</sup>Engl: jagged arrays.

```

    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 neregulat 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++)
        {
            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 imuabile (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”<sup>12</sup>. 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 cel mai frecvent pentru expresii regulate.

Exemple:

```
String caleCompleta=@"\\minimax\protect\csharp";
//ghilimelele se dubleaza intr-un verbatim string
```

---

<sup>12</sup>Engl: verbatim literals



```
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. 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`)<sup>13</sup>. 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]);
        }
    }
}
```

va afișa pe ecran:

```
Token: Oh
Token:
```

---

<sup>13</sup>A se vedea exemplele din MSDN.

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

De reținut 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 putem apela la două metode. Prima presupune folosirea unei variante supraîncărcate a metodei `Split`, în care se precizează că al doilea parametru opțiunea de ignorare a rezultatelor goale:

```
String[] tokens = s.Split( new char[]{' ', ','},
    StringSplitOptions.RemoveEmptyEntries );
```

A doua modalitate se bazează pe folosirea expresiilor regulate.

Pentru a lucra cu șiruri de caractere care permit modificarea lor (concatenări repetate, substituiri de subșiruri) se folosește clasa `StringBuilder`, din spațiul de nume `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 rulat<sup>14</sup>.

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)
    {
```

---

<sup>14</sup>Codul este scris direct în CIL.

```
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. 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 protecție, 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;
    private int myField = 42;
    public void MyMethod()
    {
        Console.WriteLine("this.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++; //gradul de acces nu permite lucrul direct cu campul
        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 asociat fiecărui obiect; câmpul stochează o valoare care contribuie la starea obiectului.

**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 eveniment<sup>1</sup> 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.

**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 este 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 poate avea parametri, nu poate avea modificatori de acces, nu este 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

Parametrii metodelor permit transmiterea de valori la apel. În general, transmiterea se face prin valoare. Acest lucru înseamnă că la apelul unei metode în stiva gestionată de compilator se copiază valoarea parametrului actual transmis, iar la revenire din metodă această valoare va fi ștearsă.

---

<sup>1</sup>Engl: event handler



Exemplificăm în listing-ul 3.1 acest lucru pentru tipul valoare; în figura 3.1 este o explicație a comportamentului.

Listing 3.1: Transmiterea de parametri de tip valoare prin valoare

```
using System;
class DemoTipValoare
{
    static void f(int b)
    {
        Console.WriteLine("la intrare in f: {0}", b );
        ++b;
        Console.WriteLine("la iesire din f: {0}", b );
    }
    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);
    }
}
```

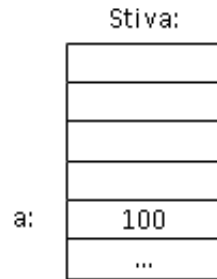
Executarea programului din listing-ul 3.1 va avea ca rezultat:

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

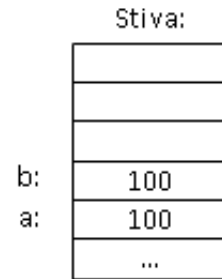
Pentru variabile de tip referință, pe stivă se depune tot o copie a valorii obiectului. Însă pentru un asemenea tip de variabilă acest lucru înseamnă că pe stivă se va depune ca valoare adresa de memorie la care este stocat obiectul respectiv. Ca atare, metoda apelată poate să modifice starea obiectului care se transmite. Codul de exemplificare este dat în listing-ul 3.2, iar exemplul este prezentat grafic în figura 3.2.

Listing 3.2: Transmiterea de parametri de tip referință prin valoare. Modificarea stării obiectului este vizibilă și după ce se iese din funcție

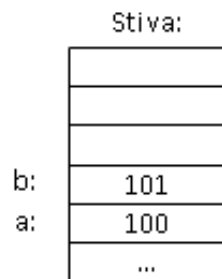
```
class Employee
{
    public String Name; //acces public pe camp pentru
    //simplificarea exemplului
    public decimal Salary; //idem
}
```



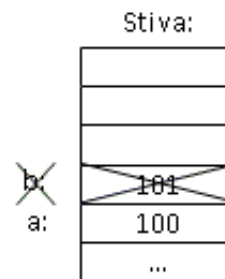
(a) Înainte de apelul metodei  $f$ : variabila locală  $a$  este alocată pe stivă și are valoarea 3.



(b) La intrarea în  $f$ : pe stivă se alocă spațiu pentru valoarea trimisă prin parametru; valoarea provine din copierea valorii parametrului actual  $a$ .



(c) După modificarea valorii lui  $b$ , înainte de ieșirea din  $f$ . Se observă că modificarea se face strict asupra valorii lui  $b$ , neafectând pe  $a$ .



(d) După ieșirea din  $f$ . Spațiul alocat pe stivă pentru parametrul  $b$  este disponibilizat, valoarea modificată în metoda  $f$  se pierde.

Figura 3.1: Transmiterea prin valoare a variabilelor de tip valoare, conform codului din listing-ul 3.1

```

}

class Test
{
    static void Main()
    {
        Employee e = new Employee();
        e.Name = "Ionescu";
        e.Salary = 300M;
        System.Console.WriteLine("inainte de apel: name={0},
                                   salary={1}", e.Name, e.Salary );
        method( e );
        System.Console.WriteLine("dupa apel: name={0}, salary={1}",
                                   e.Name, e.Salary );
    }

    static void f( Employee emp )
    {
        emp.Salary += 100M;
        emp.Name = "Ion-Ionescu";
    }
}

```

Rezultatul executării codului din listing-ul 3.2 este:

```

inainte de apel: name=Ionescu, salary=300
dupa apel: name=Ion-Ionescu, salary=400

```

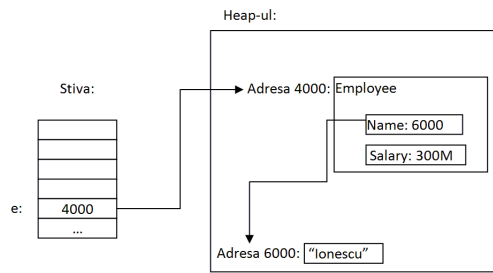
Totuși, chiar și în cazul tipului referință încercarea de a re-crea în interiorul unei metode un obiect transmis ca parametru nu are nici un efect după terminarea ei, după cum este arătat în listing-ul 3.3 și figura 3.3.

Listing 3.3: Transmitere de parametru de tip referință prin valoare. Modificarea adresei obiectului nu este vizibilă după ieșirea din funcție

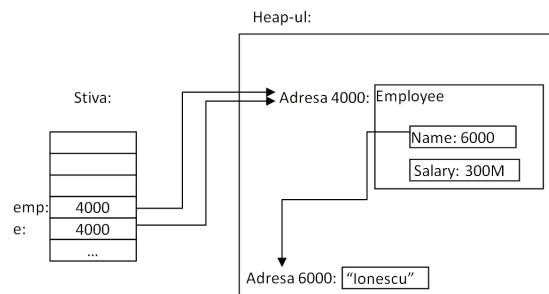
```

1 using System;
2 class MyClass
3 {
4     public int x; //camp public pentru simplitatea
5 }
6
7 class Test
8 {

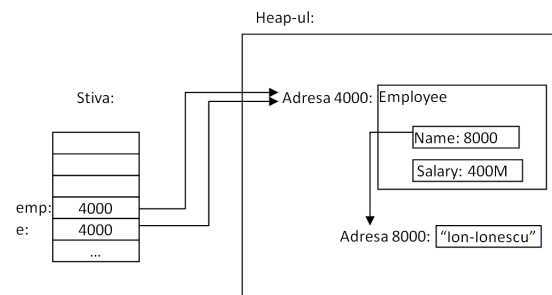
```



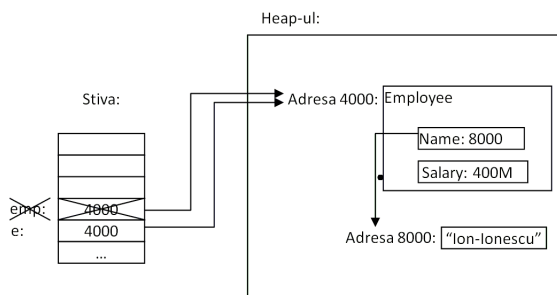
(a) Înainte de apelul metodei  $f$ : variabila locală  $e$  este alocată pe stivă și are valoarea 4000, reprezentând adresa de memorie din heap unde se găsește obiectul creat.



(b) La intrarea în  $f$ , înainte de atribuire: pe stivă se alocă spațiu pentru valoarea trimisă prin parametru; valoarea provine din copierea valorii parametrului actual  $e$ . Cele două variabile vor indica spre același obiect din heap.



(c) După atribuire, înainte de ieșirea din  $f$ . Se observă modificarea apărută în heap.



(d) După ieșirea din  $f$ . Spațiul alocat pe stivă pentru parametrul  $emp$  este disponibilizat, dar valorile din heap rămân așa cum au fost ele modificate în  $f$ .

Figura 3.2: Transmiterea prin valoare a variabilelor de tip referință, conform codului din listing-ul 3.2

```
9      static void f(MyClass b)
10     {
11         Console.WriteLine("intrare in f: {0}", b.x);
12         b = new MyClass();
13         b.x = -100;
14         Console.WriteLine("iesire din f: {0}", b.x);
15     }
16     static void Main()
17     {
18         MyClass a = new MyClass();
19         a.x = 100;
20         Console.WriteLine("inainte de apel: {0}", a.x);
21         f(a);
22         Console.WriteLine("dupa apel: {0}", a.x);
23     }
24 }
```

Ieșirea codului din listing-ul 3.3 este:

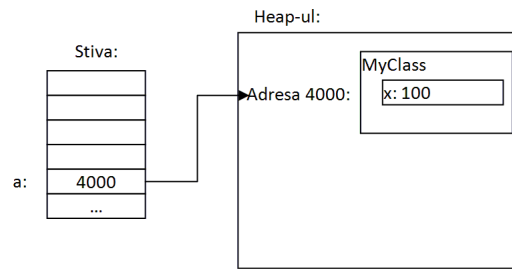
```
inainte de apel: 100
intrare in f: 100
iesire din f: -100
dupa apel: 100
```

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.

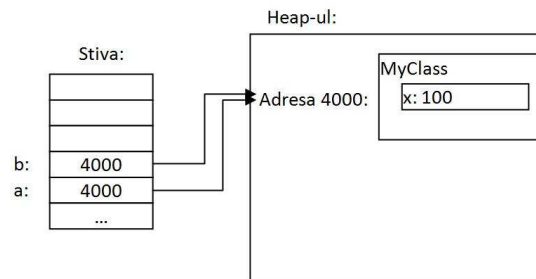
Un parametru *referință* este folosit tocmai pentru a rezolva problema transmiterii prin valoare, folosind referință (un alias) pentru entitatea dată de către metoda apelantă. Pentru a transmite un parametru prin referință, se prefixează cu cuvântul cheie *ref* la apel și la declarare de metodă, conform codului din listing-ul 3.4.

Listing 3.4: Trimitere de parametri prin referință

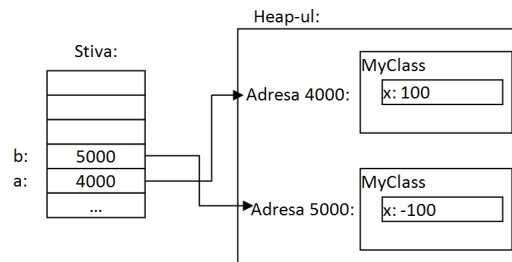
```
using System;
class Test
{
    static void swap(ref int a, ref int b)
    {
        int t = a;
        a = b;
        b = t;
    }
}
```



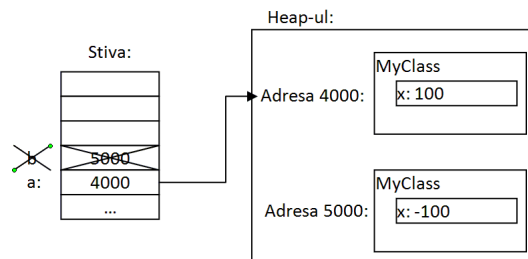
(a) Înainte de apelul metodei  $f$ : variabila locală  $a$  este alocată pe stivă și are valoarea 4000, reprezentând adresa de memorie din heap unde se găsește obiectul creat.



(b) La intrarea în  $f$ , înainte de instrucțiunea de la linia 11 din listing-ul 3.3: pe stivă se alocă spațiu pentru valoarea trimisă prin parametru; valoarea provine din copierea valorii parametru-lui actual  $a$ . Cele două variabile vor indica spre același obiect din heap.



(c) După linia 12 a aceluiași listing. Se observă modificările apărute în heap, iar  $b$  indică spre noul obiect. Obiectul  $a$  rămâne cu starea nealterată.



(d) După ieșirea din  $f$ . Spațiul alocat pe stivă pentru parametrul  $b$  este disponibilizat.

Figura 3.3: Transmiterea prin valoare a variabilelor de tip referință nu permit modificarea referinței. Figura este asociată listing-ul 3.3

```

    }

    static void Main()
    {
        int x=1, y=2;
        Console.WriteLine("inainte de apel: x={0}, y={1}", x, y);
        swap(ref x, ref y);
        Console.WriteLine("dupa apel: x={0}, y={1}", x, y);
        //se va afisa:
        //inainte de apel: x=1, y=2
        //dupa apel: x=2, y=1
    }
}

```

Explicația acestui comportament este că la apelul metodei *swap*, se trimit nu copii ale valorilor *x* și *y*, ci adresele de memorie unde se află stocate *x* și *y* pe stivă (altfel zis, referințe către *x* și *y*). Ca atare, efectul atribuirilor din cadrul metodei *swap* sunt vizibile și după ce s-a revenit din apelul ei.

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*, precum în listing-ul 3.5 duce o eroare de compilare: “Use of unassigned local variable 'x'”.

Listing 3.5: Eroare: variabilele trimise prein *ref* trebuie să fie inițializate

```

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 ieșire<sup>2</sup>, simi-

---

<sup>2</sup>În original: output parameters.

lari cu parametrii referință, cu deosebirea că nu trebuie asignată o valoare parametrului de apel înainte de apelul funcției, dar neapărat funcția trebuie să asigneze o valoare înainte de terminarea execuției ei. Un exemplu este dat în listing-ul 3.6.

Listing 3.6: Utilizarea de parametri de ieșire

```
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 )
    {
        area = l * l;
    }
}
```

Pentru toate tipurile de parametri de mai sus există o corespondență de 1 la 1 între parametrii actuali și cei formali. Un parametru vector<sup>3</sup> 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. Un exemplu este dat în listing-ul 3.7; se remarcă modalitățile de apel diferit.

Listing 3.7: Exemplu de utilizare de parametru de tip vector

```
using System;
class Test
{
    static void f(params int [] args)
    {
        Console.WriteLine("numarul de parametri: {0}", args.Length)
    }
}
```

---

<sup>3</sup>În original: parameter array.



```

        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 forma<sup>4</sup>:

```

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

```

## 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 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

---

<sup>4</sup>A se vedea [MSDN](#).

- 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 atribuiri. Conversiile implicite predefinite nu determină niciodată apariția de excepții.

### Conversiile indentitate

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 un alt tip de date, aflat deasupra în ierarhie. Astfel, se poate face conversie de la o variabilă de tip enumerare sau de tip structură către tipul `object` sau `System.ValueType` sau către o interfață pe care structura o 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 [7].

### 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 conversie 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 [7], 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  $A_E$  al  $B_E$
- 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 obiect 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 3.4: la linia 1, se declară și se inițializează o variabilă de tip `valoare`, 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-o convenție explicită, ca în linia a treia.

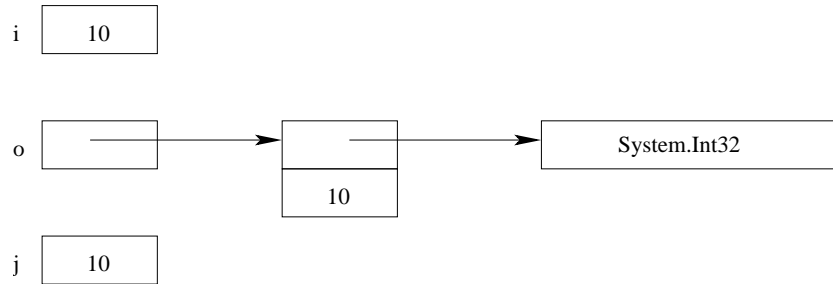


Figura 3.4: Boxing și unboxing

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 înglobată în obiect, 10.

### 3.4 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.

## 3.5 Instrucțiuni C#

### 3.5.1 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
{
    private int f(int x)
    {
        if (x >= 0) goto myLabel;
        x = -x;
        myLabel: return x;
    }
    static void Main()
    {
        DemoLabel dl = new DemoLabel();
        dl.f(-14);
    }
}
```

### 3.5.2 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 va trebui să fie terminată cu 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*.

```
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;//neaparat, 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).

### 3.5.3 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 = r;
}
```

### 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).

### 3.5.4 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 "spagetti 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 apelate de către funcția care a apelantă. 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.

### 3.5.5 Instrucțiunile try, throw, catch, finally

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

### 3.5.6 Instrucțiunile checked și unchecked

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

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

Verificare se va face la run-time.

### 3.5.7 Instrucțiunea lock

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

```
lock (x) instructiune
```

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

### 3.5.8 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( StreamWriter w = File.CreateText("log.txt") )
        {
            w.WriteLine("This is line 1");
            w.WriteLine("This is line 2");
        }
    }
}
```

## 3.6 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 nume<sup>5</sup> 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.

---

<sup>5</sup>engl: namespaces

### 3.6.1 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.



### 3.6.2 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 să apară 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;
    //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
}
```

## 3.7 Declararea unei clase

Declararea unei clase se face în felul următor:

`attributeopt 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 imbricate, cât și pentru clase care sunt conținute în spații de nume;

**internal** - se poate folosi atât pentru clase imbricate, 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 imbricate; 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 imbricate; semnifică acces limitat la clasa conținătoare; este modificatorul implicit;

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

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

**sealed** - o clasă sealed nu poate fi moștenită; poate fi clasă imbricată sau nu;

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

**partial** - clasa este definită în mai multe fișiere

### 3.8 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.

### 3.9 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<sub>opt</sub> modificatori-de-constructor declarator-de-constructor corp-constructor

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

nume-clasa (lista-parametrilor-formali<sub>opt</sub>) initializator-de-constructor<sub>opt</sub>

unde initializatorul-de-constructor are forma:

: base( lista-argumente<sub>opt</sub>) sau

: this( lista-argumente<sub>opt</sub>).

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 parametrii 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.

### 3.10 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 aibă gradul de accesibilitate cel puțin egal 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
{
    private int a;//acces implicit de tip privat
    static void Main()
    {
        A objA = new A();
        objA.a = 1;//se poate accesa in interiorul clasei
    }
}
```

### 3.10.1 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 unui instanțe de clase se poate folosi cuvântul *this*, reprezentând referință la obiectul curent.

### 3.10.2 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 se face doar prin intermediul numelui de clasă:

```
class B
{
    public static int V = 3;
    static void Main()
    {
```



```
B.V++; //corect
V++; //corect
B b = new B();
b.V++; //eroare de compilare
}
}
```

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

### 3.10.3 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 sau prin intermediul unui constructor. Valoarea unor astfel de câmpuri nu e obligatorie a fi cunoscute la compilare.

### 3.10.4 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 rulare<sup>6</sup> 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.

### 3.10.5 Inițializarea câmpurilor

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

- numeric: 0
- bool: false
- char: ‘\0’
- enum: 0
- referință: null
- structură: apel de constructor implicit

## 3.11 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. Exemplu:

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

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

---

<sup>6</sup>Engl: runtime system

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).

## 3.12 Metode

O metodă este un membru care implementează 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.

### 3.12.1 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 (de exemplu de către o metodă statică, dar se poate și dintr-o clasă imbricată — a se vedea secțiunea dedicată 4.5).

O metodă nestatică nu are cuvântul “static” specificat; ea este apelabilă doar pornind de la o referință la un obiect.

### 3.12.2 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);
    }
}
```

## Curs 4

# Clase (continuare)

### 4.1 Proprietăți

O proprietate este un membru care permite acces la partea de stare a unei clase. Exemple de proprietăți sunt: lungimea unui șir, numele unui client, textul conținut într-un control `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 `get/set` pentru clase sau obiecte.

Declararea unei proprietăți se face astfel:

`modificator-de-proprietateopt tip numeproprietate definitie-getopt definitie-setopt`  
unde `modificator-de-proprietate` este: `attributeopt modificator-de-accesopt get corp-get`  
`attributeopt modificator-de-accesopt set corp-set`

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

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

**proprietate write–only**, dacă este prezent doar accesoriul *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 accesoriu 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 accesoriul 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 *get/set*, ci și pentru scrierea controalelor .NET utilizator.

În figura 4.1 este dată reprezentarea unui control utilizator:

Codul corespunzător este dat mai jos:

```
using System;
using System.Collections;
```

Figura 4.1: Control definit de utilizator

```

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”.

## 4.2 Indexatori

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

Declararea unui indexator se face în felul următor:

atribute<sub>opt</sub> modificali-de-indexator<sub>opt</sub> declarator-de-indexator {declaratii-de-accesori}

Modificatorii de indexator pot fi: *new*, *public*, *protected*, *internal*, *private*, *protected internal*, *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 aibă vreun parametru de tip *ref* sau *out*. Declarațiile de accesori vor conține accesori *get* sau accesori *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 v.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;
        }
    }

    private 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 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.



## 4.3 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 transmiși prin valoare;
3. Același modificali nu poate apărea de mai multe ori în antetul unui operator

### 4.3.1 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 utilizare de forma:

```
if( a )
```

sau pentru cicluri `do`, `while` și `for`, precum și în operatorul ternar `"? :"`.

Exemplu:

```
public class DBBool
{
    private int value;
```

```

    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 contine 4x0, iv1 contine 4x1
        iv2 = ++iv1; // iv2 contine 4x2, iv1 contine 4x2
    }
}

```

### 4.3.2 Operatori binari

Declararea unui operator binar se face astfel:  
tip operator operator-binari-supraincercabil ( tip identificator, tip identifica-  
tor) corp

Operatorii binari supraîncercabili 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: +, -, /, \*.

Pentru supraîncărcarea operatorului de adunare pentru clasa IntVector putem scrie:

```
public class IntVector
{
    //se continua clasa IntVector scrisa mai sus
    public static IntVector operator+(IntVector a, IntVector b)
    {
        if (a == null || b == null || a.Length != b.Length)
        {
            throw new ArgumentException("Parametri neadecvati");
        }
        IntVector c = new IntVector(a.Length);
        for(int i=0; i<c.Length; i++)
        {
            c[i] = a[i] + b[i];
        }
        return c;
    }
}
```

### 4.3.3 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
{
    private 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.

#### 4.3.4 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( ));
        }
    }
}

```

## 4.4 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 modifier-de-constructor-static identificador( ) corp`

Modificatorii de constructori statici se pot da sub forma:

`externopt static` sau

`static externopt`

Constructorii statici nu se moștenesc, 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(byte red, byte green, byte blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
    private byte red;
    private byte green;
    private byte 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;
    }
}
```

## 4.5 Clase imbricate

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 LinkedList
{
    // 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 imbricate se face prin *NumeClasaExterioara.NumeClasaInterioara* (așa cum este arătat în Exemplul 1), de unde se deduce că o clasă imbricată 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 imbricate nu trebuie să fie precedată de crearea unei instanțe a clasei exterioare conținătoare, așa cum se vede din Exemplul 1. O clasă imbricată nu are vreo relație specială cu membrul predefinit *this* al clasei conținătoare. Altfel spus, nu se poate folosi *this* în interiorul unei clase imbricate pentru a accesa membri instanță din tipul conținător. Dacă o clasă imbricată 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();
    }

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

```
    }  
}
```

Se observă cu această ocazie că o clasă imbricată poate manipula toți membrii din 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 imbricate se folosesc intens în cadrul containerilor pentru care trebuie să se construiască un enumerator. Clasa imbricată va fi în acest caz strâns legată de container și va duce la o implementare ușor de urmărit și de întreținut.

## 4.6 Destructori

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

Acest mecanism de garbage collection scutește programatorul de grija dealocării memoriei. Dar există situații în care se dorește să se facă management manual al dealocării resurselor (de exemplu al resurselor care țin de sistemul de operare sau de servere: fișiere, conexiuni la rețea sau la serverul de baze 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 suprascrive sau să apeleze această metodă.

Exemplu:

```
~MyClass()
{
    // Dealocare de resurse
}
```

Metoda de mai sus este automat translatată în:

```
protected override void Finalize()
{
    try
    {
        // Dealocare de resurse
    }
    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 metodă 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 a destructorului (care am văzut că e de fapt un finalizator, metoda *Finalize*) 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 indicat ca tipurile care implementează această metodă să definească de asemenea destructor. În caz contrar, este posibil ca resursele să nu se dea loc (leak).

Exemplu:

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

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

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

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

Pentru cele obținute mai sus, 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()
```

## 4.7 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 [4.9](#))
3. conține doar membri statici

Exemplu:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        double celsius = Double.Parse(temperatureCelsius);
        double fahrenheit = celsius * 1.6 + 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;  
        }  
    }  
}
```

## 4.8 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.

### 4.8.1 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*.



### 4.8.2 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.

### 4.8.3 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
}
```

## 4.9 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.

## Curs 5

# Clase - polimorfism, clase abstracte. Structuri, interfețe, delegați

### 5.1 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.

#### 5.1.1 Polimorfismul parametric

Este cea mai slabă formă de polimorfism, fiind regăsită în majoritatea limbajelor. Prin polimorfismul parametric se permite ca o implementare de funcție să poată prelucra orice număr de parametri. Acest lucru se poate obține prin folosirea în C# a unui parametru trimis prin *params* (a se vedea secțiunea 3.2).

#### 5.1.2 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 sau numere diferite de parametri formali. Alegerea funcției care va fi apelată se va face la compilare, pe baza corespondenței între tipurile parametrilor de apel și tipurile parametrilor formali.

### 5.1.3 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 apeluri către metode având același nume și aceiași 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
        p.Draw();
        s.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 următor, doar va duce la dispariția avertismentului). Codul va afișa:

```

Polygon.Draw()
Shape.Draw()

```

Dacă prima linie afișată este conformă cu intuiția, cea de-a doua este discutabilă, dar de fapt este perfect justificată: apelul de metodă *Draw()* este rezolvat pentru ambele apeluri de mai sus 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 pentru care se 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 obiectului *s* la momentul rulării. Cu alte cuvinte, apelul ar trebui să fie rezolvat la rulare și nu la compilare, în funcție de natura efectivă a obiectelor. Acest comportament polimorfic este referit sub denumirea *polimorfism de moștenire*.

#### 5.1.4 *Virtual și override*

Pentru a asigura faptul că legarea apelului la o metodă de metodă care va fi executată 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 : Shape{
    public override void Draw(){...}
}
```

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

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

asta însemnând că s-a apelat metoda corespunzătoare “tipului efectiv” 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();
```

### 5.1.5 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 polimorfică a ei, ci apare ca o nouă metodă. Este ca și cum metoda declarată *new* ar avea nume diferit și nu se mai sesizează încercare de suprascriere.

Dacă nu se dorește suprascrierea polimorfică și nu se folosește *new*, se obține avertismentul de compilare pomenit mai sus iar suprascrierea rămâne încă nepolimorfică.

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 creată inițial de compania A. Este de dorit în astfel de cazuri compilatorul să avertizeze despre posibile nepotriviri semantice și suprascrieri – posibil accidentale. Oricum, programatorii din B vor trebui să pună specificatorul *new* înaintea metodei *B.M()* pentru a elimina avertismentul.

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. Să presupunem în continuare că se specifică *new*.

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 suprascrierii polimorfice).

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 semnala faptul că metoda *B.N()* suprascrie polimorfic 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 polimorfic o metodă virtuală.

### 5.1.6 Metode *sealed*

O metodă declarată *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*.

### 5.1.7 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 foo()* care determină afișarea clasei în care este definită și pentru care se vor specifica *new*, *virtual*, *override*, *sealed*. Să

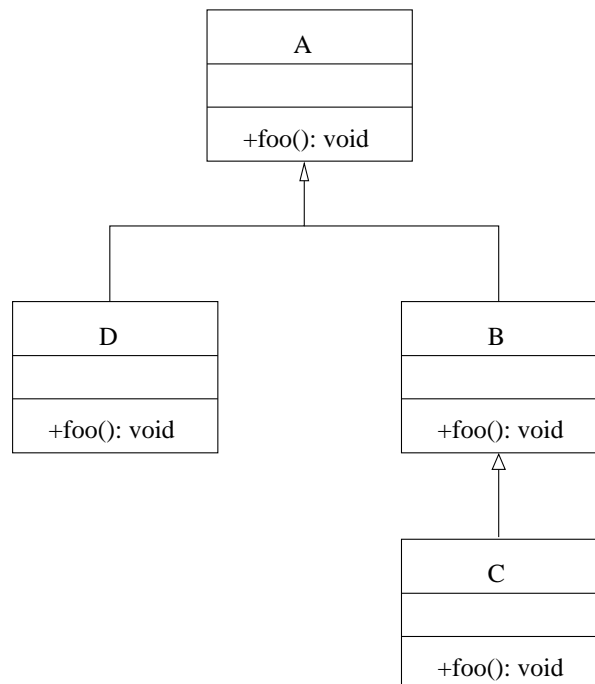


Figura 5.1: Ierarhie de clase

presupunem 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].foo();
}
/* secventa 2 */
a.foo();
b.foo();
c.foo();
d.foo();
}
}

```

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				

Tabelul 5.1 (continuare)

Metoda	A.f()	B.f()	C.f()	D.f()
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 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				

## 5.2 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 clasei. În exemplele de anterior, 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 clasei. 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. O metodă 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. O clasă abstractă nu poate fi instanțiată direct de către programator.

### 5.3 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.

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 imbricate 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 reunite 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 { }
```

## 5.4 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 modificatori-de-structopt struct identificador :interfețeopt corp ;opt`  
 Modificatorii 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). Nu poate fi supertip pentru alte tipuri de date, poate 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 imbricate. Nu poate conține destructor.

La atribuire, se face o copiere a valorilor conținute de către sursă în destinație (indiferent de natura 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.ToString(),yVal.ToString()));
    }
    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* 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 setată inițial:

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 întotdeauna 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 câmpurilor de tip structură ai unei clase.

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:

```
//bloc de instructiuni
{
    Point p;//variabila locala
    Console.WriteLine(p);//nota: aici se face boxing
}
```

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 încearcă 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 (în funcție de context) la un overhead datorat conversiei.

### 5.4.1 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* 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 și ar putea duce la fragmentarea heap-ului.

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 (consum suplimentar de memorie și cicli procesor). 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.

## 5.5 Interfețe

O interfață definește un contract comportamental. 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. Declararea unei interfețe se face astfel:

attribute<sub>opt</sub> modificali-de-interfață<sub>opt</sub> **interface** identificator baza-interfeței<sub>opt</sub> corp-interfață ;<sub>opt</sub>

Modificatorii 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 membru 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, sau să declare metodele din interfață ca fiind abstracte. 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 CSVFile : TextFile
{

```

```
public override void Read()
{
    Console.WriteLine("CSVFile.Read()");
}
public new void Write()
{
    Console.WriteLine("CSVFile.Write()");
}
}
public class Test
{
    static void Main()
    {
        Console.WriteLine("\nTextFile reference to CSVFile");
        TextFile textRef = new CSVFile();
        textRef.Read();
        textRef.Write();
        Console.WriteLine("\nISavable reference to CSVFile");
        ISavable savableRef = textRef as ISavable;
        if(savableRef != null)
        {
            savableRef.Read();
            savableRef.Write();
        }
        Console.WriteLine("\nCSVFile reference to CSVFile");
        CSVFile csvRef = textRef as CSVFile;
        if(csvRef!= null)
        {
            csvRef.Read();
            csvRef.Write();
        }
    }
}
```

La ieșire se va afișa:

```
TextFile reference to CSVFile
CSVFile.Read()
TextFile.Write()
```

```
ISavable reference to CSVFile
CSVFile.Read()
```

```
TextFile.Write()
```

```
CSVFile reference to CSVFile
```

```
CSVFile.Read()
```

```
CSVFile.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 = textRef as ISavable;
    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*, deci 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()//metoda privata!
{
    //...
}
}
```

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 în clasă/structură 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 explicita de IDataBound
    void IDataBound.Bind()
    {
        Console.WriteLine("Binding to data store...");
    }
}

class NameHidingApp
{
    public static void Main()
    {
        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 = 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 explicit care metodă (mai exact: din ce interfață) 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 2!" ) ;
}
```

atunci se poate face apel `dog.GreetOwner()` (variabila `dog` este instanță de `Dog`); apelurile de metode din interfață rămân de asemenea valide. Rezultatul este afișarea mesajului `Woof 2`.

### 5.5.1 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.

Un dezavantaj al claselor abstracte este că nu poate fi decât bază unică pentru orice altă clasă. Partea bună este că pot conține cod definit care se moștenește.

## 5.6 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 apăsător, 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:

`atributeopt modificali-de-delegatopt delegate tip-retur identificator( lista-param-formaliopt);`

Modificatorul de delegat poate fi: *new*, *public*, *protected*, *internal*, *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 (asemănător cu statutul claselor imbricate).

Exemplu:

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

### 5.6.1 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*. 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( ));
}
}

```

### 5.6.2 Delegați statici

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 [2] este dată și o implementare de delegat ca proprietate statică, prin care se creează un delegat doar în cazul în care este nevoie de el<sup>1</sup>.

---

<sup>1</sup>Tehnică numită inițializare târzie (lazy initialization)

## Curs 6

# Metode anonime. Evenimente. Excepții.

### 6.1 Metode anonime

Pentru a folosi un delegat a fost nevoie până acum de a se crea de fiecare dată o metodă (și posibil și o nouă clasă care să conțină această metodă). Există însă cazuri în care corpul metodei este suficient de simplu pentru a nu necesita declararea explicită a metodei. C# 2.0 introduce această facilități prin intermediul metodelor anonime.

Varianța tradițională presupunea scrierea unui cod de forma:

```
class SomeClass
{
    delegate void SomeDelegate();
    public void InvokeMethod()
    {
        SomeDelegate del = new SomeDelegate(someMethod);
        del();
    }
    private void someMethod()
    {
        MessageBox.Show("Hello");
    }
}
```

Se poate defini o implementare folosind o metodă anonimă:

```
class SomeClass
{
```



```
delegate void SomeDelegate();
public void InvokeMethod()
{
    SomeDelegate del = delegate()
    {
        MessageBox.Show("Hello");
    };

    del();
}
}
```

Metoda anonimă este definită in-line și nu ca metodă membră a unei clase. Compilatorul este suficient de inteligent pentru a infera tipul delegatului, pe baza declarației de variabilă delegat (del în exemplul anterior).

O astfel de metodă anonimă se poate folosi oriunde este nevoie de o variabilă de tip delegat, de exemplu ca parametru al unei metode:

```
class SomeClass
{
    delegate void SomeDelegate();

    public void MyMethod()
    {
        invokeDelegate(delegate(){MessageBox.Show("Hello");});
    }

    private void invokeDelegate(SomeDelegate del)
    {
        del();
    }
}
```

Există și cazuri în care se cere transmiterea de parametri metodei anonime. Parametrii (tip + nume) se declară în interiorul parantezelor cuvântului delegat:

```
class SomeClass
{
    delegate void SomeDelegate(string str);
    public void InvokeMethod()
    {
        SomeDelegate del = delegate(string str)
```

```

        {
            MessageBox.Show(str);
        };
    del("Hello");
}
}

```

Dacă se omit cu totul parantezele de după cuvântul `delegate`, atunci se declară o metodă anonimă care este asignabilă unui delegat cu orice semnătură:

```

class SomeClass
{
    delegate void SomeDelegate(string str);
    public void InvokeMethod()
    {
        SomeDelegate del = delegate
        {
            MessageBox.Show("Salut");
        };
        del("Parametru ignorat");
    }
}

```

Remarcăm că încă trebuie dați parametri delegatului ce se apelează; dacă delegatul declară parametri transmiși cu “out”, atunci varianta de mai sus nu se poate aplica.

### 6.1.1 Multicasting

Uneori este nevoie ca un delegat să poată apela mai mult de o singură metodă. De exemplu, atunci când un buton este apăsător, se poate să vrei să efectuezi mai mult de o sigură acțiune: să scrii într-un textbox un șir de caractere și să înregistrezi într-un fișier faptul că s-a apăsător 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. Mult mai simplu ar fi dacă unui delegat i s-ar putea atribui mai multe metode. Acest lucru se numește *multicasting* și este mecanism esențial pentru tratarea evenimentelor.

Orice delegat care returnează *void* este un delegat multicast, care poate fi tratat și ca un delegat uzual. 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 tipul variabilei *transmitter* este compatibil cu *myMulticastDelegate* (adică are aceeași semnătură). Operatorul `-=` funcționează invers față de `+=`: șterge metode din lista de apel.

Exemplu:

```
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 myMulticastDelegate
myMulticastDelegate = writer + logger;
//apeleaza 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 delegat
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 ieşire 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

todo: de facut paralela intre modalitatea primitiva/trista de implementare de la [Dimecasts](#), “Learning the Observer Pattern”, vs. delegati “Learning the Observer Pattern w/ Callbacks”.

Interfeţele grafice actuale cer ca un anumit program să răspundă la *evenimente*. Un eveniment poate fi de exemplu apăsarea unui buton, terminarea transferului unui fişier, 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 apariţia unuia se va cere reacţie din partea sistemului soft.

Alte clase pot fi interesate în a răspunde la aceste evenimente. Modul în care vor reacţiona va fi particular, iar obiectul care semnalează evenimentul (ex: un obiect de tip buton, la apăsarea lui) nu trebuie să ştie modul în care se va răspunde. Butonul va comunica faptul că a fost apăsăat, iar clasele interesate în acest eveniment vor reacţiona în consecinţă.

### 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*<sup>1</sup> deoarece publică evenimentul *Click* iar celelalte clase sunt numite *abonați*<sup>2</sup> 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 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. Tipul de retur *void* este obligatoriu, parametrii formali sunt însă tradiționali.

Declararea unui eveniment se face astfel:

`attributeopt modificali-de-evenimentopt event tip nume-eveniment`  
 Modificator-de-eveniment poate fi *abstract*, *new*, *public*, *protected*, *internal*, *private*, *static*, *virtual*, *sealed*, *override*, *extern*. *Tip* este un handler de eveniment, adică 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 );
```

---

<sup>1</sup>Engl: publisher.

<sup>2</sup>Engl: subscribers.

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 transmisă:

```

public class DisplayClock
{
    public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(timeHasChanged);
    }
    private 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
private 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 subscris 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
    private 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 ieșire se va afișa:

```

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 au nevoie să știe despre modul în care lucrează clasa *Clock*, iar clasa *Clock* nu are nevoie să știe nimic 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 multe 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 semnala 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 corectată 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<sup>3</sup>.

### 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* (deci poate fi și tip derivat din *Exception*) 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 o legătură 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

---

<sup>3</sup>Și nu neapărat a întregului proces!

- *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 detalii despre metoda care a aruncat excepția curentă<sup>4</sup>

### 6.3.2 Aruncarea și prinderea excepțiilor

#### Aruncarea cu *throw*

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

```
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...");
    }
}
```

---

<sup>4</sup>MethodBase este o clasă care pune la dispoziție informații despre metodele și constructorii unei clase

```

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

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, dar se poate face 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 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( );
    }

    //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;
    }

    //incearca sa imparta doua numere
    public void TestFunc( )
    {
        try
        {
```

```

        double a = 5;
        double b = 0;
        Console.WriteLine (“{0} / {1} = {2}”, a.ToString(), b.ToString(), 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!”);
    }
}
}

```

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 : 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 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 : 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(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 (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.Write("\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-ului<sup>5</sup>

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,

---

<sup>5</sup>Total Cost of Ownership.

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 neînlocuit.

### 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

# Colecții și clase generice

### 7.1 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. Clasele din acest spațiu de nume reprezintă containere de elemente de tip *Object* care își gestionează singure necesarul de memorie (cresc pe măsură ce se adaugă elemente; pot de asemenea să își reducă efectivul de memorie alocat atunci când numărul de elemente conținute este prea mic).

Exemplu:

```
ArrayList myCollection = new ArrayList();  
myCollection.Add(client);  
client = myCollection[0] as Client;
```

Remarcăm conversia explicită pentru recuperarea unui element de tip *Client* 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* : *IList*, *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*, *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, adică un *iterator*.

### **ICollection**

Interfața *ICollection* este tipul de bază pentru orice clasă de tip colecție; extinde interfața *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 o 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

### 7.1.1 Iteratori pentru colecții

Colecțiile (atât cele de tip listă, cât și cele dictionar) implementează interfața *IEnumerable* care permite construirea unui obiect de tip enumerator instanță a lui *IEnumerator*:

```
interface IEnumerator {  
    object Current {get;}  
    bool MoveNext();  
    void Reset();  
}
```

Remarcăm că un asemenea iterator permite citirea doar înainte a datelor din colecția peste care iterează. 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: vom 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 esențial pentru funcționarea instrucțiunii *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);  
}
```

Deoarece proprietatea `Current` este read-only, putem justifica acum cele spuse în secțiunea 3.5.3 pentru instrucțiunea `foreach` nu permite modificarea valorii variabilei cu care se face iterarea. Pentru o modificare ar fi trebuit ca să fie prezent și accesorul `set`.

### 7.1.2 Colecții de tip listă

Colecțiile de tip listă sunt: *ArrayList*, *BitArray*, *Stack*, *Queue* și *CollectionBase*.

#### ArrayList

Este o clasă concretă (i.e. instanțiabilă) 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, folosind un indice de poziție:

```
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 booleani, 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 tip colecție ce implementează politica *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.

### 7.1.3 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 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("cheie: {0}, valoare: {1}", e.Key, e.Value);
```

fie implicit:

```
foreach (DictionaryEntry s in htable)
    Console.WriteLine("cheie: {0}, valoare: {1}", s.Key, s.Value);
```

### 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).

## 7.2 Crearea unei colecții

Vom exemplifica în această secțiune modul în care se definește o colecție ce poate fi iterată. Sunt prezentate 2 variante. În ambele cazuri clasa de tip colecție va implementa interfața *IEnumerable*, dar va diferi modul de implementare.

### 7.2.1 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 tipul imbricat *MyEnumerator* primește prin constructor o referință la obiectul de tip colecție, deoarece orice clasă imbricată în C# este automat și statică, neavând astfel acces la membrii nestatici ai clasei.

Demonstrația pentru iterarea clasei este:

```
class DemoIterator
{
    static void Main()
    {
        MyCollection col = new MyCollection();
        foreach(int s in col)
        {
            Console.WriteLine(s.ToString());
        }
    }
}
```

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 un design pattern), abordarea presupune scrierea multor linii de cod. Programatorii evită această variană de implementare, preferând mecanisme alternative precum indexatorii, ceea ce poate duce la spargerea încapsulării.
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 heap și ciclul procesor consumați vor afecta 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 imbricate (ele fiind clase auxiliare), ceea ce încalcă principiul încapsulării. 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.

### 7.2.2 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 semnaliza valoarea ce urmează a fi returnată sau oprirea iterării. Are formele:

```
yield return expresie;  
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 iterare 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*.

Ca să facem evident acest mecanism de continuare a execuției de la punctul de retur anterior, considerăm exemplul:

```
using System;  
using System.Collections.Generic;  
  
namespace Iterator  
{  
    class DemoCorutina  
    {  
        static IEnumerable<int> Numere()  
        {  
            Console.WriteLine("Returnare 1");  
            yield return 1;  
            Console.WriteLine("Returnare 2");  
            yield return 2;  
            Console.WriteLine("Returnare 3");  
            yield return 3;  
        }  
  
        static void Main(string[] args)  
        {  
            foreach(int valoare in Numere())  
            {  
                Console.WriteLine(valoare.ToString());  
            }  
        }  
    }  
}
```

```
}
```

pentru care rezultatul afișat pe ecran este:

```
Returnare 1
1
Returnare 2
2
Returnare 3
3
```

deci în mod clar apelul pentru următoarea valoare dată de către metoda `Numere` se continuă de la punctul de ieșire anterior. De fapt, compilatorul va genera automat o implementare de metodă de tip *IEnumerable* (precum am făcut manual în secțiunea 7.2.1), permițându-se astfel programatorului să se concentreze pe designul metodei și mai puțin pe stufoasele detalii interne. Un alt aspect demn de reținut este că secvența se construiește pe măsură ce datele din enumerare sunt parcurse.

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 iterare peste nodurile din 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 popularea arborelui cu valori
        public void AddValues(params T[] value)
        {
            Array.ForEach(value, Add);
        }
        #endregion

        #region tehnici de traversare
        public IEnumerable<T> InOrder()
        {
            return inOrder(root);
        }

        public IEnumerable<T> PreOrder()
        {
            return preOrder(root);
        }
        #endregion

        #region Private helper methods
        private IEnumerable<T> inOrder(TreeNode<T> node)
        {
            if (node.Left != null)
            {
                foreach (T value in inOrder(node.Left))
                {
                    yield return value;
                }
            }
            yield return node.Value;
        }
    }
}
```

```

        if (node.Right != null)
        {
            foreach (T value in inOrder(node.Right))
            {
                yield return value;
            }
        }
    }

private IEnumerable<T> preOrder(TreeNode<T> root)
{
    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 parcureri (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.

## 7.3 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 scriere unui cod general, ce poate fi ulterior adaptat automat la cerințele specifice ale programatorului.

### 7.3.1 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, nu tratează decât tipurile de date cunoscute.
2. scrierea unei metode care să folosească un *Object* pe post de tip al parametrilor; 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 specificare 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 al metodei generice. Cel puțin unul din parametrii formali însă trebuie să fie de tip  $T$ .

### 7.3.2 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, 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.

### 7.3.3 Constrângeri asupra parametrilor de genericitate

Pentru structura de mai sus este posibil să se folosească o declarație de forma:

```

Point<StringBuilder> r;

```

ceea ce este aberant din punct de vedere semantic. Am dori să putem face restricționarea cât mai mult a 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.ValueType</i> (să fie tip valoare)
<i>where T:class</i>	<i>T</i> trebuie să nu fie derivat din <i>System.ValueType</i> (să fie tip referință)
<i>where T:new()</i>	<i>T</i> trebuie să aibă un constructor implicit (fără parametri)
<i>where T:NameOfBaseClass</i>	<i>T</i> trebuie să fie derivat (direct sau nu) din <i>NameOfBaseClass</i> sau chiar tipul <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>...
```

### 7.3.4 Interfețe și delegați generici

Interfețele și delegații pot fi declarați ca fiind generici; deși nu prezintă cerințe sau particularități față de ceea ce s-a spus mai sus, le evidențiem deoarece gradul înalt de abstractizare le face utile în modelarea orientată pe obiecte.

```
interface IMyFeature<T>
{
    T MyService(T param1, T param2);
}
```

respectiv:

```
delegate void MyGenericDelegate<T>(T arg);
```

## 7.4 Colecții generice

### 7.4.1 Probleme cu colecțiile de obiecte

Colecțiile, așa cum au fost ele prezentate în secțiunea 7.1 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 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 greșeala.



### 7.4.2 Colecții generice

Clasele generice împreună cu colecțiile au fost combinate în biblioteca .NET Framework, ducând la apariția unui nou spațiu de nume, în *System.Collections.Generic*. Acesta conține tipurile: *ICollection<T>*, *IComparer<T>*, *IDictionary<K, V>*, *IEnumerable<T>*, *IEnumerator<T>*, *ICollection<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*.

### 7.4.3 Metode utile în colecții

În multe situații pentru colecții de date se cere rezolvarea de probleme des întâlnite, precum sortarea sau căutarea de elemente. Colecțiile prezintă implementări eficiente pentru algoritmi de sortare și căutare. Vom exemplifica acest lucru pentru clasa *ArrayList*, în care putem avea criterii de comparare diverse, specificate la momentul rulării.

```
class Student  
{  
    private String name;  
    private double averageGrade;  
  
    public String Name  
    {  
        get{return name;}  
        set{name = value;}  
    }  
  
    public double AverageGrade  
    {  
        get{return averageGrade;}  
        set{averageGrade = value;}  
    }  
}
```

```
/// <summary>
/// Implementeaza comparatie intre 2 studenti dupa nota
/// </summary>
class ComparerStudentGrade : IComparer<Student>
{
    #region IComparer<Student> Members

    public int Compare(Student x, Student y)
    {
        if (x.AverageGrade < y.AverageGrade)
        {
            return -1;
        }
        if (x.AverageGrade == y.AverageGrade)
        {
            return 0;
        }
        return +1;
    }

    #endregion
}

/// <summary>
/// Implementeaza comparatie intre 2 studenti dupa nume
/// </summary>
class ComparerStudentName : IComparer<Student>
{
    #region IComparer<Student> Members

    public int Compare(Student x, Student y)
    {
        return String.Compare(x.Name, y.Name);
    }

    #endregion
}

class Program
{
```

```
static void Main(string[] args)
{
    //pregatirea datelor
    Student s1 = new Student();
    s1.Name = "B"; s1.AverageGrade = 3;
    Student s2 = new Student();
    s2.Name = "A"; s2.AverageGrade = 5;
    Student s3 = new Student();
    s3.Name = "C"; s3.AverageGrade = 1;
    List<Student> students = new List<Student>();
    students.Add(s1); students.Add(s2); students.Add(s3);
    Console.WriteLine("original data");
    displayStudents(students);

    //sortare dupa nume
    students.Sort(new ComparerStudentName());
    Console.WriteLine( "sorted by name" );
    displayStudents(students);

    //sortare dupa nota
    students.Sort(new ComparerStudentGrade());
    Console.WriteLine("sorted by grade");
    displayStudents(students);

    //cautare binara intr-o colectie *deja* sortata
    //*dupa acelasi criteriu folosit pentru cautare*
    Student s4 = new Student();
    s4.Name = "B";
    ComparerStudentName myComparer = new ComparerStudentName();
    Console.WriteLine( "this student appears on position: {0}",
        students.BinarySearch( s4, myComparer).ToString() );
}

private static void displayStudents(List<Student> students)
{
    foreach (var student in students)
    {
        Console.WriteLine("{0} {1}", student.Name,
            student.AverageGrade.ToString() );
    }
}
```

}

Convenția privind valoarea returnată de metoda `Compare` este: dacă primul argument este mai mic decât al doilea, atunci valoare negativă; dacă argumentele sunt egale, atunci 0; altfel, valoarea returnată trebuie să fie pozitivă.

Pentru situația în care se dorește sortarea descrescătoare se poate crea o nouă clasă de implementare a comparării care să schimbe semnele rezultatelor; alternativ, se poate porni de la colecție ordonată crescător și apelând metoda `Reverse()` se inversează ordinea elementelor.

## Curs 8

# ADO.NET

### 8.1 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 ocupată pentru niște operații care nu necesită neapărat conectare continuă
- probleme legate de scalabilitatea aplicației: se poate ca serverul de baze de date să lucreze eficient cu 5-10 conexiuni menținute, dar dacă numărul acestora crește serverul poate să reacționeze lent
- pentru unele servere se pot impune 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 cu baze de date decât cele precedente (e.g. ADO, ODBC).

Pentru o prezentare a metodelor de lucru cu surse de date sub platformă Windows se poate consulta [8].

Vom exemplifica în cele ce urmează preponderent pe bază de date Microsoft SQL 2008 Express Edition, ce se poate descărca gratuit de pe site-ul Microsoft.

## 8.2 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 sursă de date să se folosească o bibliotecă de clase specializată. 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 programării orientate pe obiecte).

Există următorii furnizori de date<sup>1</sup> (lista nu este completă):

Tabelul 8.1: 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 Sever 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
PostgreSQL	Npgsql	Server de baze de date PostgreSQL
MySql	MySql	SGBD MySql

Prefixele trecute în coloana a doua sunt folosite pentru clasele de lucru specifice unui anumit furnizor ADO.NET: de exemplu, pentru o conexiune SQL Server se va folosi clasa *SqlConnection*.

---

<sup>1</sup>Engl: Data Providers.

## 8.3 Componentele unui furnizor de date

Fiecare furnizor de date ADO.NET oferă în principal patru componente: *Connection*, *Command*, *DataReader*, *DataAdapter*. Arhitectura ADO.NET este prezentată în figura 8.1

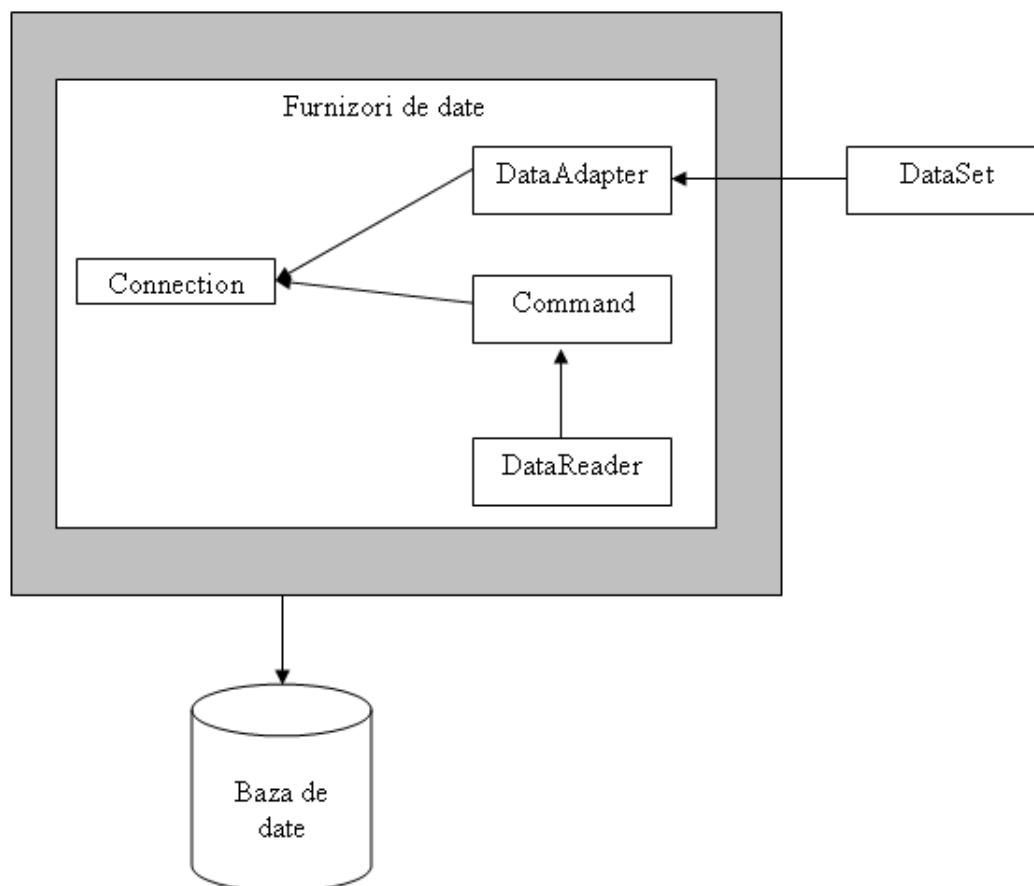


Figura 8.1: Principalele clase ADO.NET

Mai jos sunt descrieri succinte ale claselor cel mai des utilizate.

### 8.3.1 Clasele *Connection*

Sunt folosite pentru a reprezenta o conexiune la sursa de date. 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 tranzații sau setarea perioadei de time-out. Stau la baza oricărei accesări de servicii de pe server.

### 8.3.2 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*. Pot fi folosite pentru apelarea de proceduri stocate aflate pe server. Ele permit scrierea de interogări SQL parametrizate sau specificarea parametrilor pentru procedurile stocate.

### 8.3.3 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 rapid și cu minim de resurse consumate.

### 8.3.4 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 prelucrarea deconectată a datelor și ulterior reflectarea modificărilor pe baza de date. Conțin 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.

### 8.3.5 Clasa *DataSet*

Această clasă nu este parte a unui furnizor de date .NET ci independentă de particularitățile de conectare și lucru cu o sursă 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.

## 8.4 Obiecte *Connection*

Clasele de tip *Connection* pun la dispoziție tot ceea ce e necesar pentru conectarea la baze de date. Este primul tip de date cu care un programator ia contact atunci când vrea să folosească un furnizor de date .NET. Înainte



ca o comandă să fie executată pe o bază de date trebuie stabilite datele de conectare și deschisă conexiunea.

Orice clasă de tip conexiune (din orice furnizor de date) implementează interfața *IDbConnection*. Înțelegem deci că ambele clase *SqlConnection* (folosită pentru conectare la server Microsoft SQL Server 2000) și *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 SQL ("localhost") precum și al numelui de instanță pentru acest server ("express"), 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*.

### 8.4.1 Proprietăți

1. *ConnectionString*: de tip *String*, cu accesorii *get* și *set*; această proprietate definește un string 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.
  - *Data Source* (sinonim cu *server*): 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*): specifica 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 *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");//30 de secunde
```

Se poate specifica pentru *Connect Timeout* valoarea 0 cu semnificația “așteaptă oricât”, dar se recomandă 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. E 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 numele furnizorului 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*.

### 8.4.2 Metode

1. *Open()*: deschide o conexiune la baza de date
2. *Close()*, *Dispose()*: închid conexiunea
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.

### 8.4.3 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.

### 8.4.4 Stocarea stringului de conexiune în fișier de configurare

Este contraindicat ca stringul de conexiune să fie scris direct în cod; modificarea datelor de conectare (de exemplu parola pe cont sau locația sursei de date) ar necesita recompilarea codului.

.NET Framework permite menținerea într-un fișier a unor perechi de tipul cheie—valoare, specifice aplicației. 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. Elementul rădăcină împreună cu declarația de XML sunt:

```
<?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\squlexpress;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 eaceastă clasă: din Solution 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();
    //a se vedea mai jos cum se asigura inchiderea conexiunii
  }
}
```

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\squlexpress;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;
```

#### 8.4.5 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 conectare. Dacă dimensiunea maximă setată a grupării este atinsă, atunci nu se va mai crea o conexiune nouă, ci se va pune cererea într-o coadă de așteptare. Dacă așteptarea durează mai mult decât este precizat în valoarea proprietății 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 gruparea de conexiuni, degrevându-l pe programator de această aspect ce nu ține de logica aplicației. La dorință comportamentul implicit se poate modifica prin intermediul conținutului stringului de conectare.

#### 8.4.6 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:

```
IDBConnection con = ...  
try
```

---

<sup>2</sup>Engl: connection pooling

```
{
    //deschidere conexiune
    //lucru pe sursa de date
}
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. În același scop se mai poate folosi și instrucțiunea `using` (secțiunea 3.5.8), deoarece orice clasă de tip conexiune implementează interfața `IDisposable`:

```
using(IDBConnection con = ...)
{
    //deschidere conexiune
    //lucru pe baza
}
//aici se executa garantat con.Dispose()
//si deci se inchide conexiunea
```

## 8.5 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.ConnectionStrings["constring"]
    .ConnectionString;
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.

### 8.5.1 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.IDbConnection*, 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 8.1).
5. *Parameters* - proprietate de tip *IDbCommandParameters*, 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 în cadrul căreia se execută comanda curentă.

### 8.5.2 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 și afișează 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:

```
using(SqlConnection con = new SqlConnection(
    ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString))
{
    SqlCommand cmd = new SqlCommand();
    cmd.CommandText = @"DELETE FROM Customers
    WHERE CustomerID = 'SEVEN'";
    cmd.Connection = con;
    con.Open();
```



```
    Console.WriteLine(cmd.ExecuteNonQuery().ToString());
} // automat se apeleaza con.Dispose(), care inchide conexiunea
```

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 tabeli Customers într-un obiect de tip *SqlDataReader* (se presupune că baza de date se stochează pe un server MSSQL):

```
using(SqlConnection con = new SqlConnection(
    ConfigurationManager.ConnectionStrings["constring"]
        .ConnectionString))
{
    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();
} // automat se apeleaza con.Dispose(), care inchide conexiunea
```

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 metadate 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();
```

### 8.5.3 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 aibă proprietatea *CommandType* la valoarea *CommandType.StoredProcedure* iar proprietatea *CommandText* să conțină numele procedurii stocate:

```
using(SqlConnection con = new SqlConnection(
    ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString))
{
    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();
} //automat se disponibilizeaza obiectul de conexiune,
//deci si conexiunea la baza de date

```

*Observație:* fiecare conexiune se poate 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 nu va putea fi închisă.

#### 8.5.4 Folosirea comenzilor parametrizate

Există posibilitatea de a rula cod SQL (interogări sau 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".

```

using(SqlConnection con = new SqlConnection(
    ConfigurationManager.ConnectionStrings["constring"]
        .ConnectionString))
{
    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();
} //se disponibilizeaza obiectul de conexiune

```

Pentru parametrul creat s-a setat tipul lui (ca fiind tip șir de caractere SQL) și valoarea. De reținut faptul că numele parametrului se prefixează cu caracterul “@” în cazul lucrului cu SQL Server.

În cazul în care un parametru este de ieșire, acest lucru trebuie spus explicit folosind proprietatea *Direction* a parametrului respectiv:

```

using(SqlConnection con = new SqlConnection(
    ConfigurationManager.ConnectionStrings["constring"]
        .ConnectionString))
{
    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());
} //se inchide automat conexiunea

```

Remarcăm următoarele:

- este posibil ca într-o comandă să se execute mai multe interogări
- pentru parametrul de ieșire numit “@count” trebuie făcută declarație de direcție; implicit un parametru este de intrare
- parametrii de ieșire sunt accesibili doar după închiderea obiectului de tip *DataReader*

## 8.6 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 interfața *IDataReader*
2. se lucrează conectat la sursa de date - pe toată perioada cât este accesat un obiect *DataReader* necesită conexiune activă
3. este read-only; dacă se dorește modificarea datelor se poate folosi un *DataSet* + *DataAdapter* sau comenzi INSERT, DELETE sau UPDATE trimise prin obiect de tip *Command*;
4. este forward-only - metoda de modificare a poziției curente este doar în direcția înainte; orice reîntoarcere presupune reluarea citirii.

Avantajele utilizării acestui tip de obiecte sunt: accesul conectat, performanțele bune, consumul mic de resurse și tipizarea puternică.

### 8.6.1 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

### 8.6.2 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 tipuri 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(int index)* - returnează *true* dacă în câmpul specificat prin index este o valoare de *NULL* în 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 înainte de primele înregistrări.

### 8.6.3 Crearea și utilizarea unui *DataReader*

Nu se poate crea un obiect de tip *DataReader* prin apel de constructor, ci prin intermediul unui obiect de tip *Command*, folosind apelul *ExecuteReader()* (a se vedea secțiunea 8.3.2). Pentru comanda respectivă se specifică instrucțiunea care determină returnarea setului de date precum și obiectul de conexiune. Această conexiune trebuie să fie deschisă înainte de apelul *ExecuteReader()*. Trecerea la următoarea înregistrare se face folosind metoda

*Read()*. 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:

```
using(SqlConnection conn = new SqlConnection (
    ConfigurationManager.ConnectionStrings["constring"]
        .ConnectionString))
{
    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 ();
}
```

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 disponibilă cât mai devreme pentru a putea fi reutilizată.

### 8.6.4 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 de conexiuni la sursa 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 () );
```

### 8.6.5 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 putea utiliza din plin facilitățile tipului respectiv. Pentru ca acest lucru să nu se întâmple se pot folosi metodele *GetX**Y* 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 9

# ADO.NET (2)

### 9.1 Obiecte *DataAdapter*

La fel ca și *Connection*, *Command*, *DataReader*, clasa *DataAdapter* face parte din furnizorul de date .NET specific fiecărui tip de sursă de date. Scopul clasei 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;
using System.Configuration;

class DemoDataSource
{
    static void Main()
    {
        SqlConnection conn = new SqlConnection(
```

```
        ConfigurationManager.ConnectionStrings["constring"]
            .ConnectionString);
DataSet ds = new DataSet();
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*.

### 9.1.1 Metode

1. *Constructori* — de la cel implicit 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 tabelei 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, a se vedea (a))

De remarcat că un *DataAdapter* își poate deschide și închide singur conexiunea, dar dacă aceasta a fost deschisă înaintea metodei *Fill()* atunci programatorul trebuie să o închidă explicit.

3. *Update()* – metodă polimorfică, permițând reflectarea modificărilor efectuate într-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.

### 9.1.2 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 tabelului din obiectul *DataSet* în care sunt depuse. Poate avea următoarele valori:
  - *MissingSchemaAction.Add* - implicit, *DataAdapter* adaugă coloana la schema tabelului
  - *MissingSchemaAction.AddWithKey* - ca mai sus, dar adaugă și metadata relativ la cine este cheia primară
  - *MissingSchemaAction.Ignore* - se ignoră nepotrivirea dintre coloanele aduse din baza de date și cele existente în obiectul *DataSet*, ceea ce duce la pierdere de date pe *DataSet*
  - *MissingSchemaAction.Error* - se generează o excepție de tipul *InvalidOperationException*.

## 9.2 Clasa *DataSet*

Clasa *DataSet* nu mai face parte din biblioteca unui furnizor de date ADO.NET, fiind parte din .NET Framework. 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 pot apela la serverul de baze de date doar când preiau datele sau când doresc salvarea modificărilor. 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* face abstracție de sursa de date, procesarea datelor desfășurându-se independent de natura furnizorului de date.

Figura 9.1 conține o vedere parțială asupra clasei *DataSet*.

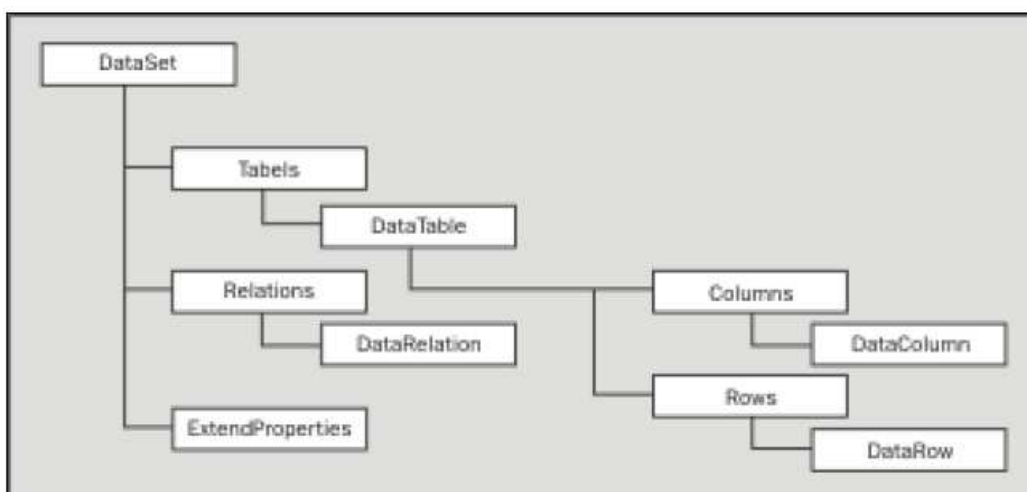


Figura 9.1: Structura unui *DataSet*

### 9.2.1 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 între tabele.
3. Colecția *ExtendedProperties* conține proprietăți definite de utilizator.

### 9.2.2 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", typeof(String));
```

după care coloana nou creată se adaugă la obiect tabel din *DataSet* cu:

```
myTable.Columns.Add(myColumn);
```

Definirea unei coloane ca fiind cu capacitate 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.AutoIncrementStep = 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["Name"] = "Book";  
tempRow["CategoryID"] = 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 a valorilor.
- *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 din alte tabele. Valorile care se pot seta pentru o asemenea constrângere se specifică în proprietățile *DeleteRule* și *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 același DataSet). Restricția se adaugă la tabela 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 inregistrarea

```

### 9.2.3 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 tablouri 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"]);

```

### 9.2.4 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*. Odată creat 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 tablei care va conține datele:



```
//defineste comanda de selectare din baza de date
String mySqlStmt ="SELECT * FROM Customers";
String myConnectionString = ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString;
//Construieste obiectul de conexiune + obiectul de comanda SELECT
SqlConnection myConnection = new SqlConnection(myConnectionString);
SqlCommand myCommand = new SqlCommand(mySqlStmt, myConnection);
//Construieste 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*.

### 9.2.5 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();
```

### 9.2.6 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 utilizează 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 câmpului/câmpurilor cheie, iar pentru obiectul *DataAdapter* care face aducerea din bază se setează proprietatea *MissingSchemaAction* pe valoarea *MissingSchemaAction.AddWithKey* (implicit este *Add*).

Fiecare linie modificată din colecția *Rows* a unei tabele din obiectul *DataSet* 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 pentru o tabelă din baza de date.

```
SqlConnection conn = new SqlConnection(
```

```
ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString);
da = new SqlDataAdapter("SELECT id, name, address FROM
    customers", conn);
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
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 tabele, 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 din 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 primară. 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 = scope_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 calificată ca Identity (SGBD-ul este cel care face managementul valorilor acestor câmpuri, nu programatorul). *scope\_identity()* este o funcție predefinită ce returnează 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)";
```

## 9.3 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

Este inadmisibil (deși posibil) 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 ignorate

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();
//acum obiectele SqlCommand au proprietatea de conexiune setata
SqlTransaction myTrans;

myTrans = myConnection.BeginTransaction();
//Trebuie asignat obiectul de tranzactie
//celor doua comenzi care sunt in aceeasi tranzactie
myCommand1.Connection = myConnection;
myCommand2.Connection = myConnection;
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
{
    myTrans.Rollback();
}
finally
{
    myConnection.Close();
}
```

Comanda de *ROLLBACK* se poate executa și în alte situații, de exemplu comanda efectuată depășește stocul disponibil sau alte reguli de logică a aplicației.

## 9.4 Lucrul generic cu furnizori de date

În cele expuse până acum, s-a lucrat cu un furnizor de date specific pentru SQL Server 2008. 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* (în esență combinație de *Abstract factory* și *Factory Method*).

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");
    ...
}
```

Preferăm să evităm codificarea numelui furnizorului de date în cod (precum mai sus) și sugerăm specificarea lui în fișier 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\v4.0.30319\config).

Exemplu: fișierul de configurare este:

```
<configuration>
  <appSettings>
    <!-- Provider -->
    <add key="provider" value="System.Data.SqlClient" />
    <!-- String de conexiune -->
  </appSettings>
  <connectionStrings>
    <add name="cnStr" connectionString=
      "Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/>
  </connectionStrings>
</configuration>
```

Codul C#:

```
static void Main(string[] args)
{
  string dp = ConfigurationManager.AppSettings["provider"];
  string cnStr = ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString;

  DbProviderFactory df = DbProviderFactories.GetFactory(dp);
  DbConnection cn = df.CreateConnection();
  cn.ConnectionString = cnStr;
  try
  {
    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();
  }
}
```



```
finally
{
    cn.Close();
}
```

## 9.5 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;
```

Construcția `tip?` este un alias pentru `Nullable<tip>`, unde `Nullable` este o structură generică:

```
public struct Nullable<T> where T : struct, new()
```

E logic să nu putem defini ca nulabile tipurile referință, deoarece acestea suportă implicit valoare de `null`:

```
//eroare de compilare
string? s = null
```

Lucrul cu tipurile nulabile se face exact ca și cu tipurile referință:

```
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 10

## LINQ (I)

### 10.1 Elemente specifice C# 3.0

Secțiunea conține o prezentare a elementelor introduse de versiunea 3.0 a limbajului C#.

#### 10.1.1 Proprietăți implementate automat

Considerăm clasa:

```
class MyClass
{
    private int myField;

    public int MyField
    {
        get
        {
            return myField;
        }
        set
        {
            myField = value;
        }
    }
}
```

Deseori se pune problema scrierii unor câmpuri private, pentru care accesarea se face prin intermediul proprietăților. Este contraindicat să se expună

câmpurile ca fiind publice, deoarece se sparge încapsularea și nu se poate face databinding la câmpuri publice. Dacă un câmp se expune ca fiind public și un cod client începe să îl acceseze, este imposibil ca ulterior să se impună cod de validare pentru accesul la el.

Deoarece codul de tipul celui scris mai sus apare foarte des, s-a pus problema simplificării lui. În C# 3.0 se scrie echivalent:

```
class MyClass
{
    public int MyField
    {
        get;
        set;
    }
}
```

Se folosește aici mecanismul de implementare automată a unei proprietăți care acționează asupra unui câmp privat autodeclarat; această proprietate returnează sau accesează direct câmpul asociat.

Particularitățile sunt următoarele:

1. nu se declară câmpul privat; acesta este creat automat de compilator, pe baza proprietății auto-implementate;
2. nu se scriu implementări pentru `get` și `set`; corpul lor este caracterul “;”. `get` accesează câmpul autodeclarat, `set` setează valoarea câmpului cu ce se află în dreapta semnului egal;
3. nu se poate accesa câmpul autodeclarat altfel decât prin intermediul proprietății; el este de fapt anonim;
4. proprietatea nu poate fi read-only sau write-only, ci doar read–write.

Dacă ulterior se decide implementarea unui accesori de către programator, atunci și celălalt trebuie implementat, iar câmpul privat trebuie declarat în mod explicit. Important este însă că se scrie un minim de cod pentru a genera un contract: câmp privat accesat prin proprietate publică.

### 10.1.2 Inițializatori de obiecte

Să considerăm clasa:

```
class Person
{
    private string firstName;
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string lastName;
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }
    private int age;
    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

Se poate scrie următoarea secvență de cod care inițializează o persoană cu datele cuvenite:

```
Person p = new Person();
p.FirstName = "Rafael";
p.Age = 25;
p.LastName = "Popescu";
```

În C# 3.0 se poate scrie mai succint:

```
Person p = new Person { FirstName = "Rafael",
    Age = 25, LastName = "Popescu" };
```

cu același rezultat<sup>1,2</sup>. Într-un astfel de caz se face mai întâi apelarea constructorului implicit (indiferent de cine anume îl scrie - compilatorul sau

---

<sup>1</sup>În unele lucrări se folosește: `Person p = new Person(){FirstName="Rafael", Age=25, LastName="Popescu" };`. Remarcăm parantezele rotunde după numele clasei folosite de operatorul `new`.

<sup>2</sup>Cele două secvențe din text nu sunt totuși echivalente, așa cum se arată în <http://community.bartdesmet.net/blogs/bart/archive/2007/11/22/c-3-0-object-initializers-revisited.aspx>

programatorul) și apoi se face accesarea proprietăților, în ordinea scrisă la inițializator. Membrii pentru care se face inițializare trebuie să fie publici; în particular, ei pot fi și câmpuri, dar acest lucru nu este încurajat de principiul încapsulării. Putem avea inclusiv proprietăți auto-implementate (secțiunea 10.1.1).

Dacă se scrie un constructor care preia un parametru dar nu și unul care să fie implicit, de exemplu:

```
public Person(String firstName)
{
    FirstName = firstName;
}
```

atunci se poate încă folosi mecanismul de inițializare:

```
Person r = new Person("Rafael") {LastName="Popescu", Age = 25 };
```

Exemplul se poate dezvolta mai departe prin exemplificarea construirii unor obiecte mai complexe:

```
Person p = new Person{
    FirstName = "Rafael",
    LastName = "Popescu",
    Age=25,
    Address = new Address{
        City = "Brasov",
        Country = "Romania",
        Street = "Iuliu Maniu"
    }
}
```

Am presupus mai sus că avem definiție corespunzătoare a clasei *Address* și a proprietății cu același nume.

### 10.1.3 Inițializatori de colecții

Se dă secvența de cod:

```
List<String> list = new List<String>();
list.Add("a");
list.Add("b");
list.Add("c");
```

În C# 3.0 ea este echivalentă cu:

```
List<String> list = new List<String>(){ "a", "b", "c" };;
```

ceea ce aduce aminte de o trăsătură similară de la inițializarea tablourilor; mai exact codul de mai jos:

```
String[] x = new String[3];
x[0] = "a";
x[1] = "b";
x[2] = "c";
```

este încă din prima versiune de C# echivalentă cu:

```
String[] x = new String[] { "a", "b", "c" };;
```

Inițializarea colecțiilor vine să ofere același mecanism ca și în cazul tablourilor.

Exemplul poate fi completat cu popularea unei colecții de obiecte compuse:

```
List<Person> persons = new List<Person>(){
    new Person{FirstName = "Rafael", LastName="Popescu", Age=25},
    new Person{FirstName = "Ioana", LastName="Ionescu", Age=23}
};
```

#### 10.1.4 Inferența tipului

Considerăm metoda:

```
void f()
{
    int x = 3;
    MyClass y = new MyClass();
    bool z = true;
    ....
}
```

Pentru fiecare din declarațiile cu inițializare de mai sus se poate spune că valoarea asociată dă suficientă informație despre tipul de date corespunzător variabilelor. Tocmai din acest motiv începând C# 3.0 se poate scrie astfel:

```
void f()
{
    var x = 3;
    var y = new MyClass();
    var z = true;
    ....
}
```

`var` nu reprezintă un nou tip de date, ci pur și simplu arată că la compilare se poate deduce tipul actual al variabilelor locale respective. Ca atare, `var` este de fapt o scurtătură, prin care se obține ceea ce s-a scris prima oară. Dacă după declarare și inițializare se scrie:

```
x = false;
```

compilatorul semnalează eroare, deoarece s-a făcut deja inferarea tipului de dată pentru `x`, și anume `int`, iar `false` nu este compatibil cu `int`. Limbajul rămâne deci puternic tipizat, fiecare variabilă având un tip asignat.

Folosirea acestui nou cuvânt cheie se supune condițiilor:

- se poate folosi doar pentru variabile locale
- se poate folosi atunci când compilatorul poate să infereze tipul de dată asociat variabilei; acest lucru se întâmplă conform situațiilor de mai sus, sau pentru o iterare de forma:

```
int[] sir = {1, 2, 3};  
foreach(var i in sir)  
{  
    ...  
}
```

- odată ce compilatorul determină tipul de date, acesta nu mai poate fi schimbat.

Mecanismul nu reprezintă la prima vedere un mare pas, dar este cu adevărat util atunci când se lucrează cu alte mecanisme din C# 3.0: tipuri anonime și programarea bazată pe LINQ.

### 10.1.5 Tipuri anonime

Acest mecanism permite declararea unor variabile de un tip care nu este definit aprioric. Se omite declararea numelui de clasă și a componentei acesteia. Exemplu:

```
var p = new {FirstName="Rafael", Age=25};
```

Pentru proprietățile care sunt pomenite în expresia de inițializare se face deducerea tipului în mod automat (pe baza aceluiași mecanism de la variabile anonime). Proprietățile sunt publice și read-only.



### 10.1.6 Metode parțiale

Metodele parțiale reprezintă metodele care sunt declarate în mai multe părți ale unei clase. Clasa conținătoare poate să fie sau nu parțială. Cele două părți sunt una în care se definește metoda parțială (cu tip de retur și parametri, dar fără corp) și alta în care se implementează (cu corp complet).

Exemplu:

```
//fisierul MyClass1.cs
partial class MyClass
{
    //declaratie de metoda
    partial void f(int x);
}

//fisierul MyClass2.cs
partial class MyClass
{
    //implementare de metoda
    partial void f(int x)
    {
        Console.WriteLine(x.ToString());
    }
}
```

Compilatorul va pune la un loc cele două declarații de metode parțiale și va rezulta o metodă “întreagă”. Regulile care trebuie respectate pentru crearea de metode parțiale sunt:

1. metoda pentru care se folosește implementare parțială trebuie să returneze `void`;
2. parametrii nu pot fi de tip `output`;
3. metoda nu poate avea specificatori de acces; ea este privată
4. metoda trebuie să fie declarată atât la definire cât și la implementare ca fiind parțială.

Adăugăm că o astfel de metodă parțială poate să apară într-o structură sau într-o clasă (declarate ca parțiale). Nu este obligatoriu ca ambele declarații să apară în părți diferite ale clasei. Dacă o implementare de metodă parțială lipsește, atunci orice apel la ea este ignorat.

### 10.1.7 Metode de extensie

Metodele de extensie permit scrierea de metode asociate cu clase, alte clase decât cele în care sunt definite.

Să considerăm o clasă, în cazul căreia fiecare obiect menține un șir de numere. O posibilă definiție ar fi:

```
class LotOfNumbers
{
    private int[] numbers;

    public LotOfNumbers(int[] numbers)
    {
        this.numbers = new int[numbers.Length];
        numbers.CopyTo(this.numbers, 0);
    }

    public int NumbersNo
    {
        get
        {
            if (numbers == null)
            {
                return 0;
            }
            else
            {
                return numbers.Length;
            }
        }
    }

    public int Sum()
    {
        if (numbers == null)
        {
            throw new Exception("No number inside");
        }
        int result = 0;
        foreach (int x in numbers)
        {
            result += x;
        }
    }
}
```

```
    }  
    return result;  
}  
}
```

Ne propunem să adăugăm o metodă la această clasă, care să returneze media elementelor șirului. Să presupunem că nu avem acces la sursa codului C#. Singura modalitate de a extinde clasa `LotOfNumbers` este ca să se scrie o metodă de extensie:

```
static class ExtendsLotOfNumbers  
{  
    public static double Average(this LotOfNumbers data)  
    {  
        return (double)data.Sum() / data.NumbersNo;  
    }  
}
```

Utilizarea metodei de extensie se face cu:

```
class Program  
{  
    static void Main()  
    {  
        LotOfNumbers numbers = new LotOfNumbers(new int[]{1, 2, 3});  
        Console.WriteLine(numbers.Average().ToString());  
    }  
}
```

Am reușit astfel să “strecurăm” o metodă în interiorul unei clase al cărei cod sursă nu este accesibil. Putem utiliza acest mecanism dacă:

1. clasa în care se face implementarea metodei de extensie este statică;
2. metoda care implementează extensia este statică
3. metoda de extensie are primul parametru de tipul clasei pentru care se face extinderea, iar tipul parametrului formal este prefixat cu `this`.

Este posibil ca o metodă de extensie să aibă mai mult de un parametru.

## 10.2 Generalități

Language Integrated Query (LINQ, pronunțat precum “link”) permite interogarea unor colecții de date folosind o sintaxă integrată în platforma .NET. Prin intermediul unor operatori se pot interoga colecții de forma: tablouri, colecții, clase enumerabile, documente XML, baze de date relaționale. Datele rezultate sunt văzute ca obiecte; are loc o mapare (asociere, traducere) a unor date neobiectuale într-un format ușor de folosit în limbajele obiectuale din cadrul platformei. Trebuie menționat că sintaxa este unitară, independent de natura sursei de date.

Listing 10.1: Interogare LINQ

```
var query = from e in employees
where e.id == 1
select e.name;

sau:
```

Listing 10.2: Altă interogare LINQ

```
var results = from product in productsCollection
where product.UnitPrice < 100
select new {product.ProductName, product.UnitPrice};

foreach (var result in results)
{
    Console.WriteLine(result);
}
```

Există interfața `IQueryable<T>`, care permite implementarea unor furnizori de date în modul specific sursei de date considerate. Expresia folosită pentru interogare este tradusă într-un arbore de expresie. Dacă colecția implementează `IEnumerable<T>`, atunci se folosește motorul de execuție LINQ local, integrat în platformă; dacă colecția implementează `IQueryable<T>`, atunci se folosește implementarea bazată pe arborele de expresie; această implementare este dată de către furnizoare de LINQ. Furnizoarele de LINQ sunt scrise în mod specific fiecărei surse de date, dar datorită respectării unor interfețe specificate, detaliile de implementare sunt irelevante pentru cel care folosește cod LINQ în interogare.

LINQ a fost introdus în versiunea 3.5 a lui .NET Framework. Constă într-un set de unelte care sunt folosite pentru lucrul cu date și extensii aduse limbajului. Este alcătuit din:

- LINQ to Objects – se aduc date din colecții care implementează `IEnumerable<T>`; datele interogate sunt deja în memoria procesului;

- LINQ to XML – convertește documentele XML într-o colecție de obiecte de tip `XElement`;
- LINQ to SQL – permite convertirea interogărilor LINQ în comenzi SQL;
- LINQ to DataSets – spre deosebire de LINQ to SQL care a venit inițial doar cu suport pentru SQL Server, LINQ to DataSets folosește ADO.NET pentru comunicarea cu baze de date;
- LINQ to Entities – soluție Object/Relational Mapping de la Microsoft ce permite utilizarea de Entities – introduse în ADO.NET 3.0 – pentru a specifica declarativ structura obiectelor ce modelează domeniul și folosește LINQ pentru interogare.

La ora actuală există următorii furnizori de date LINQ:

1. LINQ to MySQL, PostgreSQL, Oracle, Ingres, SQLite și Microsoft SQL Server
2. LINQ to CSV
3. LINQ to Google
4. LINQ to NHibernate
5. LINQ to System Search

Este dezvoltat și PLINQ, Parallel LINQ, un motor ce folosește paralelizarea de cod pentru executare mai rapidă a interogărilor, în cazul unui sistem multinucleu sau multiprocesor.

## 10.3 Motivație

Vom prezenta două motive pentru care LINQ este util:

- codul stufos, neproductiv utilizat pentru accesarea în modul clasic a datelor;
- nepotrivirea paradigmelor obiectual-relaționale.

### 10.3.1 Codul clasic ADO.NET

Pentru accesarea datelor dintr-o bază de date relațională, folosind ADO.NET se scrie de regulă un cod de forma:

Listing 10.3: Interogare clasică folosind ADO.NET

```
using (SqlConnection connection = new SqlConnection("..."))
{
    connection.Open();
    SqlCommand command = connection.CreateCommand();
    command.CommandText =
        @"SELECT Name, Country
        FROM Customers
        WHERE City = @City";
    command.Parameters.AddWithValue("@City", "Paris");
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            string name = reader.GetString(0);
            string country = reader.GetString(1);
            ...
        }
    }
}
```

Se remarcă următoarele:

- cantitatea de cod scrisă; codul de sus este unul des folosit, scrierea lui în repetate rânduri este contraproductivă;
- interogările sunt exprimate prin intermediul unei fraze scrise între ghilimele, ca șir de caractere, deci automat nu se poate verifica prin compilarea corectitudinea codului SQL conținut; este adevărat, însă, că se pot folosi aici proceduri stocate care sunt compilate deja pe server;
- slaba tipizare a parametrilor: dacă tipul acestora nu coincide cu ce se află în baza de date? dacă numărul de parametri este incorect (asta se semnalează numai la rulare, ca eroare; de preferat ar fi fost să se depisteze acest lucru la compilare);
- de cele mai multe ori trebui folosit un dialect de SQL specific producătorului serverului de baze de date; codul SQL nu este portabil; totodată: mixarea de limbaje – C# și SQL – face codul greu de urmărit.

O expresie LINQ care demonstrează depășirea acestor probleme este:

Listing 10.4: Cod LINQ

```
from customer in customers
where customer.Name.StartsWith("A") &&
    customer.Orders.Count > 10
orderby customer.Name
select new { customer.Name, customer.Orders }
```

### 10.3.2 Nepotrivirea de paradigme

Paradigma este “o construcție mentală larg acceptată, care oferă unei comunități sau unei societăți pe perioadă îndelungată o bază pentru crearea unei identități de sine (a activității de cercetare de exemplu) și astfel pentru rezolvarea unor probleme sau sarcini”<sup>3</sup>.

Există o diferență sesizabilă între programarea orientată pe obiecte, utilizată în cadrul limbajelor folosite pentru implementarea aplicațiilor și modul de stocare și reprezentare a datelor: XML sau baze de date relaționale. Traducerea unui graf de obiecte din reprezentarea obiectuală într-o altă reprezentare este greoaie: programatorul trebuie să înțeleagă și particularitățile structurilor de date folosite pentru persistarea lor, pe lângă cunoașterea limbajului în care lucrează.

Problema pe care LINQ o abordează este considerarea următoarelor inegalități:

- “Data != Objects”
- “Relational data != Objects”
- “XML data != Objects”
- “XML data != Relational data”

Toate aceste nepotriviri necesită efort de adaptare din partea programatorului. Modelarea obiectual-relațională este problema cea mai des întâlnită, cu următoarele aspecte:

1. tipurile de date folosite de către modelele relaționale și modelele obiectuale nu sunt aceleași; de exemplu, multitudinea de tipuri șir de caractere folosite în specificarea coloanelor, în timp ce în .NET există doar tipul `String`;

---

<sup>3</sup>Definiția [Wikipedia](#).

2. modelele relaționale folosesc normalizarea (pentru eliminarea redundanței și a anomaliilor de inserare, ștergere, modificare), în timp ce modelarea obiectuală nu trebuie să treacă prin așa ceva; în schimb, modelarea obiectuală folosește agregarea sau moștenirea, mecanisme care nu își au un echivalent direct în modelarea relațională
3. modele de programare diferite: pentru SQL se folosește un limbaj declarativ care specifică *ce* se dorește, în timp ce limbajele de programare folosite sunt de regulă imperative – arată *cum* se face prelucrarea
4. încapsulare – ascunderea detaliilor și legarea laolaltă a datelor cu metodele care prelucrează datele;

Toate aceste probleme se manifestă începând cu determinarea corespondențelor între obiecte și datele persistate. Aceeași problemă apare dacă ne referim la XML, care favorizează un model ierarhic, semistructurat. Programatorul trebuie să scrie mereu un cod care să faciliteze legarea acestor universuri diferite. LINQ vine cu o propunere de rezolvare.

## 10.4 LINQ to Objects: exemplificare

LINQ to Objects este folosit pentru interogarea datelor care se află deja în memorie. Un prim exemplu este:

```
using System;
using System.Linq;
static class HelloWorld
{
    static void Main()
    {
        string[] words =
        { "hello", "wonderful", "linq", "beautiful", "world" };
        var longWords =
            from word in words
            where word.Length >= 5
            select word;
        foreach (var word in shortWords)
        {
            Console.WriteLine(word);
        }
    }
}
```



Un exemplu mai complex este:

Listing 10.5: LINQ peste colecție generică, folosind expresie LINQ

```
List<Person> people = new List<Person> {
    new Person() { ID = 1,
        IDRole = 1,
        LastName = "Anderson",
        FirstName = "Brad"
    },
    new Person() {
        ID = 2,
        IDRole = 2,
        LastName = "Gray",
        FirstName = "Tom"
    }
};
var query =
    from p in people
    where p.ID == 1
    select new { p.FirstName, p.LastName };
```

Interogarea din final poate fi scrisă și altfel:

Listing 10.6: LINQ peste colecție generică, folosind apeluri de metode

```
var query = people
    .Where(p => p.ID == 1)
    .Select(p => new { p.FirstName, p.LastName } );
```

### 10.4.1 Expresii lambda

Aceste expresii simplifică scrierea delegaților și a metodelor anonime. Într-un exemplu anterior s-a folosit:

```
Where(p => p.ID == 1)
```

care s-ar citi: “obiectul `p` produce expresia logică `p.ID` egal cu 1”. Ar putea fi rescrisă echivalent, astfel:

Listing 10.7: Expresie lambda rescrisă cu delegați

```
Func<Person, bool> filter = delegate(Person p)
{ return p.ID == 1; };
var query = people
    .Where(filter)
    .Select(p => new { p.FirstName, p.LastName } );
```

Tipul de date *Func* se declară astfel:

```
public delegate TResult Func<in T, out TResult>(
    T arg
)
```

(despre semnificația lui *in* și *out* se va discuta în secțiunea 13.6).

## 10.5 Operatori LINQ

Tabelul 10.1: Operatori LINQ.

Operație	Operator	Descriere
Agregare	<b>Aggregate</b>	Aplică o funcție peste o secvență
	<b>Average</b>	Calculează media peste o secvență
	<b>Count/LongCount</b>	calculează numărul de elemente dintr-o secvență
	<b>Max, Min, Sum</b>	Calculează maximul, minimul, suma elementelor dintr-o secvență
Concatenare	<b>Concat</b>	Concatenează elementele a două secvențe
Conversie	<b>AsEnumerable</b>	Converteste o secvență la un <b>IEnumerable&lt;T&gt;</b>
	<b>AsQueryable</b>	Converteste o secvență la un <b>IQueryable&lt;T&gt;</b>
	<b>Cast</b>	converteste un element al unei secvențe la un tip specificat
	<b>OfType</b>	Filtrează elementele unei secvențe, returnându-le doar pe cele care au un tip specificat
	<b>ToArray</b>	transformă în vector
	<b>ToDictionary</b>	transformă în dicționar
	<b>ToList</b>	transformă în colecție
	<b>ToLookup</b>	creează un obiect de tip <b>Lookup&lt;K, T&gt;</b> dintr-o secvență
	<b>ToSequence</b>	returnează argumentul transformat într-un <b>IEnumerable&lt;T&gt;</b>
Obținere de element	<b>DefaultIfEmpty</b>	dă o valoare implicită dacă secvența este goală

Tabelul 10.1 (continuare)

Operație	Operator	Descriere
	<code>ElementAt</code>	returnează elementul de la poziția specificată
	<code>ElementAtOrDefault</code>	returnează elementul de la poziția specificată sau o valoare implicită, dacă la poziția specificată nu se află nimic
	<code>First, Last</code>	primul, respectiv ultimul element dintr-o secvență
	<code>FirstOrDefault</code> <code>LastOrDefault</code>	ca mai sus, dar cu returnare de valoare implicită dacă primul, respectiv ultimul element din colecție nu este disponibil
	<code>Single</code>	returnează elementul din colecție, presupusă a fi format dintr-un singur element
	<code>SingleOrDefault</code>	ca mai sus, sau element implicit dacă elementul singular nu este găsit în secvență
Egalitate	<code>SequenceEqual</code>	verifică dacă două secvențe sunt egale
Generare	<code>Empty</code>	returnează o secvență goală de tipul specificat
	<code>Range</code>	Generează o secvență de numere aflate între două capete specificate
	<code>Repeat</code>	Generează o secvență prin repetarea de un număr de ori specificat a unei valori specificate
Grupare	<code>GroupBy</code>	Grupează elementele unei secvențe
Joncțiune	<code>GroupJoin</code>	O joncțiune grupată a două secvențe pe baza unor chei care se potrivesc
	<code>Join</code>	Joncțiune interioară a două secvențe
Sortare	<code>OrderBy</code>	Ordonează elementele unei secvențe pe baza unuia sau a mai multor criterii
	<code>OrderByDescending</code>	Ca mai sus, dar cu sortare descrescătoare
	<code>Reverse</code>	inversează ordinea elementelor dintr-o secvență
	<code>ThenBy</code> , <code>ThenByDescending</code>	pentru specificarea de chei suplimentare de sortare
Partiționare	<code>Skip</code>	Produce elementele unei secvențe care se află după o anumită poziție
	<code>SkipWhile</code>	Sare peste elementele unei secvențe care îndeplinesc o anumită condiție
	<code>Take</code>	Ia primele elemente dintr-o secvență
	<code>TakeWhile</code>	Ia elementele de la începutul unei secvențe, atâta timp cât ele respectă o

Tabelul 10.1 (continuare)

Operație	Operator	Descriere
		anumită condiție
Proiecție	Select	definește elementele care se iau în secvență
	SelectMany	Preia elemente dintr-o secvență conținând secvențe
Cuantificatori	All	Verifică dacă toate elementele unei secvențe satisfac o condiție dată
	Any	Verifică dacă vreun elementul al unei secvențe satisface o condiție dată
	Contains	Verifică dacă o secvență conține un element
Restricție	Where	Filtrează o secvență pe baza unei condiții
Mulțime	Distinct	Returnează elementele distincte dintr-o colecție
	Except	Efectuează diferența a două secvențe
	Intersect	Returnează intersecția a două mulțimi
	Union	Produce reuniunea a două secvențe

# Curs 11

## LINQ (II)

### 11.1 LINQ to Objects

Exemplele de mai jos sunt preluate din [5] și sunt folosite pentru exemplificarea codului ce folosește LINQ. Se pleacă de la clase `Person`, `Role` și `Salary`, definite ca mai jos:

Listing 11.1: Clase pentru exemplificarea LINQ-ului

```
class Person
{
    public int ID
    {
        get;
        set;
    }

    public int IDRole
    {
        get;
        set;
    }

    public string LastName
    {
        get;
        set;
    }

    public string FirstName
```

```
{
    get;
    set;
}
}

class Role
{
    public int ID
    {
        get;
        set;
    }

    public string RoleDescription
    {
        get;
        set;
    }
}

class Salary
{
    public int IDPerson
    {
        get;
        set;
    }

    public int Year
    {
        get;
        set;
    }

    public double SalaryYear
    {
        get;
        set;
    }
}
```

### 11.1.1 Filtrarea cu Where

Pentru operatorul **Where** s-au definit două metode de extensie:

```
public static IEnumerable<T> Where<T>(  
    this IEnumerable<T> source, Func<T, bool> predicate);  
  
public static IEnumerable<T> Where<T>(  
    this IEnumerable<T> source, Func<T, int, bool> predicate);
```

Prima formă folosește un predicat (condiție) care pentru un obiect de tipul **T** returnează un boolean, iar în al doilea caz se folosește la condiție obiectul și indicele său în secvență.

Să presupunem că avem o colecție de obiecte de tip **Person** construită astfel:

```
List<Person> people = new List<Person> {  
    new Person {  
        ID = 1,  
        IDRole = 1,  
        LastName = "Anderson",  
        FirstName = "Brad"  
    },  
    new Person {  
        ID = 2,  
        IDRole = 2,  
        LastName = "Gray",  
        FirstName = "Tom"  
    },  
    new Person {  
        ID = 3,  
        IDRole = 2,  
        LastName = "Grant",  
        FirstName = "Mary"  
    },  
    new Person {  
        ID = 4,  
        IDRole = 3,  
        LastName = "Cops",  
        FirstName = "Gary"  
    }  
};
```

Obținerea obiectelor de tip `Person` care au prenumele “Brad” se face cu *expresia LINQ*:

```
var query = from p in people
             where p.FirstName == "Brad"
             select p;
```

Elementele care sunt aduse de către interogarea anterioară pot fi iterate cu:

```
foreach (Person x in query)
{
    Console.WriteLine("{0}, {1}", x.FirstName, x.LastName);
}
```

Pentru expresia de interogare de mai sus se poate scrie echivalent:

```
var query = people.Where(p => p.FirstName == "Brad");
```

adică o variantă în care se folosesc metodele de extensie.

Dacă se dorește folosirea în cadrul condiției de filtrare a poziției elementului curent în lista `people`, se poate scrie:

```
var query = people.Where((p, index) => p.IDRole == 1
    && index % 2 == 0);
```

și utilizarea metodei de extensie este singura posibilitate de a referi indicele de poziție.

### 11.1.2 Operatorul de proiecție

Pentru a determina ce conține fiecare element care compune o secvență, se folosește operatorul `Select`, definit de metodele de extensie:

```
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source, Func<T, S> selector);

public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source, Func<T, int, S> selector);
```

Selectarea se poate face cu:

```
var query = from p in people
             select p.FirstName;
```

sau:

```
var query = people.Select(p => p.FirstName);
```

Se poate de asemenea ca la fiecare element selectat să se producă un obiect de un tip nou, chiar de tip anonim:



```
var query = people
    .Select(
        (p,index) => new {
            Position=index,
            p.FirstName,
            p.LastName
        }
    );
```

### 11.1.3 Operatorul let

**Let** permite evaluarea unei expresii și atribuirea rezultatului către o variabilă care poate fi utilizată în cadrul interogării ce o conține.

```
static void Main(string[] args)
{
    // code from DevCurry.com
    var arr = new[] { 5, 3, 4, 2, 6, 7 };
    var sq = from num in arr
              let square = num * num
              where square > 10
              select new { num, square };

    foreach (var a in sq)
        Console.WriteLine(a);

    Console.ReadLine();
}
```

Rezultat:

```
{ num = 5, square = 25 }
{ num = 4, square = 16 }
{ num = 6, square = 36 }
{ num = 7, square = 49 }
```

### 11.1.4 Schimbarea ordinii

Sunt folosiți operatorii: **OrderBy**, **OrderByDescending**, **ThenBy**, **ThenByDescending** și **Reverse**.

Operatorii `OrderBy` și `OrderByDescending` produc sortarea crescătoare, respectiv descrescătoare a unor secvențe, pe baza unei chei de sortare. Exemplu:

```
var q = from m in typeof(int).GetMethods()
        orderby m.Name
        select new { Name = m.Name };
```

Folosind metode se poate scrie:

```
q = typeof(int).
    GetMethods().
    OrderBy(method => method.Name).
    Select(method => new { Name = method.Name });
```

Pentru sortarea descrescătoare expresia LINQ este:

```
var q = from m in typeof(int).GetMethods()
        orderby m.Name descending
        select new { Name = m.Name };
```

sau prin operatori LINQ:

```
q = typeof(int).
    GetMethods().
    OrderByDescending(method => method.Name).
    Select(method => new { Name = method.Name });
```

Dacă se dorește specificarea mai multor chei după care să se facă sortarea în ordine lexicografică, atunci se poate specifica prin `ThenBy` și `ThenByDescending` – apelate ca metode – care sunt criteriile suplimentare de sortare. Dacă se folosește expresie LINQ, atunci se poate scrie mai simplu:

```
var query = from p in people
            orderby p.FirstName, p.LastName
            select p;
```

Exprimarea echivalentă cu operatori LINQ este:

```
var query = people.OrderBy(p => p.FirstName)
                  .ThenBy(p => p.LastName);
```

Pentru personalizarea sortării, se poate folosi o implementare de interfață `IComparer<T>`.

Inversarea ordinii elementelor dintr-o secvență se face cu `Reverse`, apelată ca metodă:

```
var q = (from m in typeof(int).GetMethods()
        select new { Name = m.Name }).Reverse();
```

sau:

```
var q = typeof(int)
    .GetMethods()
    .Select(method => new { Name = method.Name })
    .Reverse();
```

### 11.1.5 Partiționare

Se folosesc pentru extragerea unei anumite părți dintr-o secvență. Operatorii sunt: `Take`, `Skip`, `TakeWhile`, `SkipWhile`. Vom exemplifica doar o parte din aceștia:

```
//int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int[] numbers = Enumerable.Range(1, 10).ToArray();
var query = numbers.Take(5); //secventa 1, 2, 3, 4, 5
var query2 = numbers.Skip(5); //secventa 6, 7, 8, 9

var query3 = numbers.TakeWhile((n, index) => n + index < 4);
//produce: 1, 2

var query4 = numbers.SkipWhile((n, index) => n + index < 4);
//produce: 3, 4, ...9
```

### 11.1.6 Concatenarea

Se face cu `Concat`, care preia două secvențe și produce o a treia, reprezentând concatenarea lor, fără eliminarea duplicatelor:

```
int[] x = { 1, 2, 3 };
int[] y = { 3, 4, 5 };
var concat = x.Concat(y); //concat = 1, 2, 3, 3, 4, 5
```

### 11.1.7 Referirea de elemente din secvențe

Se pot folosi: `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`, `ElementAt`, `ElementAtOrDefault` și `DefaultIfEmpty`.

Pentru `First` și `FirstOrDefault` lui există formele:

```
public static T First<T>(
    this IEnumerable<T> source);
```

```

public static T First<T>(
    this IEnumerable<T> source ,
    Func<T, bool> predicate );

public static T FirstOrDefault<T>(
    this IEnumerable<T> source );

public static T FirstOrDefault<T>(
    this IEnumerable<T> source ,
    Func<T, bool> predicate );

```

Dacă nu se specifică predicatul, atunci se va returna primul element din secvență, altfel primul element din secvență care satisface condiția exprimată prin predicat.

```

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.First();
query = numbers.First(n => n % 2 == 0);

```

Varianta cu `OrDefault` este utilă dacă se crede că predicatul poate să nu fie satisfăcut de niciun element al secvenței. Se returnează valoarea implicită pentru tipul de date considerat. Dacă se folosește varianta fără `OrDefault`, atunci e posibil ca să se arunce excepție – asta în cazul în care niciun element al colecției nu respectă condiția.

### 11.1.8 Operatorul `SelectMany`

Are metodele de extensie:

```

public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source ,
    Func<T, IEnumerable<S>> selector );

public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source ,
    Func<T, int , IEnumerable<S>> selector );

```

Prin `SelectMany` se iterează peste o colecție  $X$  care conține ca elemente colecții  $X1, X2 \dots$  și se creează o lista conținând toate elementele din  $X1$  urmate de cele din  $X2$  etc.

Exemplu: pentru numerele de la 100 la 200 dorim să aflăm lista tuturor divizorilor lor (chiar cu duplicate). Asta înseamnă că dorim să obținem lista divizorilor lui 100 (*i.e.* 1, 2, 4, 5, 10, 20, 25, 50, 100) urmată de lista

divizorilor lui 101 (1, 101) etc, finalizând cu lista divizorilor lui 200. Pentru a afla divizorii unui întreg vom folosi o metodă de extensie pentru tipul `int`:

```
static class MyExtensionMethods
{
    //naive code
    public static IEnumerable<int> Divisors(this int n)
    {
        for (int i = 1; i <= n; i++)
        {
            if (n % i == 0)
            {
                yield return i;
            }
        }
    }
}
```

(avem și posibilitatea de a scrie mai scurt corpul metodei `Divisors` astfel:

```
return (from i in Enumerable.Range(1, n) where n%i == 0 select i).ToList();)
```

iar codul pe care îl încercăm în primă fază este:

```
var numbers = Enumerable.Range(100, 101);
var list = from n in numbers
           select n.Divisors();
```

Problema cu această abordare este că fiecare element al lui `list` este la rândul lui o colecție de elemente:

```
foreach (var divisorsList in list)
{
    foreach (var y in divisorsList)
    {
        Console.WriteLine("{0}, ", y);
    }
    Console.WriteLine();
}
```

iar noi am vrea să obținem direct o listă de numere și nu o listă de liste. Pentru aceasta, în primă fază rescriem interogarea pentru `list` folosind operatori LINQ:

```
var list = numbers.Select(n => n.Divisors());
```

iar pentru ca din *listă de liste de numere* să obținem direct *listă de numere*, se folosește metoda **SelectMany** în loc de **Select**<sup>1</sup>:

```
var list = numbers.SelectMany(n => n.Divisors());
iar afișarea se face iterând peste fiecare număr din lista de numere list:
foreach (var divisor in list)
{
    Console.WriteLine(divisor.ToString());
}
```

Alt exemplu: să presupunem că specificăm niște obiecte rol:

```
List<Role> roles = new List<Role> {
    new Role { ID = 1, RoleDescription = "Manager" },
    new Role { ID = 2, RoleDescription = "Developer" }
};
```

Dacă dorim rolul persoanelor având id-ul mai mare ca 1, atunci putem scrie:

```
var query = from p in people
    where p.ID > 1
    from r in roles
    where r.ID == p.IDRole
    select new
    {
        p.FirstName,
        p.LastName,
        r.RoleDescription
    };
};
```

Echivalent, se poate folosi metoda **SelectMany**:

```
var query = people
    .Where(p => p.ID > 1)
    .SelectMany(p => roles
        .Where(r => r.ID == p.RoleID)
        .Select(r => new {
            p.FirstName,
            p.LastName,
            r.RoleDescription
        })
    );
```

---

<sup>1</sup>A se vedea și excelenta explicație vizuală de la [A Visual Look At The LINQ SelectMany Operator](#).

`SelectMany` permite gestiunea unei alte secvențe elemente; dacă s-ar folosi `Select` în loc de `SelectMany`, atunci s-ar obține o secvență de elemente de tip `IEnumerable<T>`.

### 11.1.9 Joncțiuni

Operatorul `Join` poate fi folosit pentru a aduce date din două colecții, date care sunt puse în corespondență pe baza unei egalități de valori. Se bazează pe metoda de extensie:

```
public static IEnumerable<V> Join<T, U, K, V>(
    this IEnumerable<T> outer ,
    IEnumerable<U> inner ,
    Func<T, K> outerKeySelector ,
    Func<U, K> innerKeySelector ,
    Func<T, U, V> resultSelector );
```

De exemplu, pentru a aduce lista numelor, prenumelor și a descrierilor de roluri pentru colecțiile `persons` și `roles` putem folosi:

```
var query = from p in people
             join r in roles
             on p.IDRole equals r.ID
             select new {
                 p.FirstName ,
                 p.LastName ,
                 r.RoleDescription
             };
```

Folosind operatori (metode) LINQ, se poate scrie:

```
query = people.Join(roles , p => p.IDRole , role => role.ID ,
    (p, r) => new {
        p.FirstName ,
        p.LastName ,
        r.RoleDescription
    }
);
```

### 11.1.10 Grupare

Gruparea datelor se face cu `GroupBy`; se grupează elementele unei secvențe pe baza unui selector.

Exemplu: plecând de la

```
var query = from m in typeof(int).GetMethods()
            select m.Name;
```

Dacă se afișează elementele din colecția `query`, se poate observa că metoda `ToString` este afișată de 4 ori, `Equals` de două ori etc. Gruparea acestora după nume se poate face astfel:

```
var q = from m in typeof(int).GetMethods()
        group m by m.Name into gb
        select new {Name = gb.Key};
```

Dacă dorim să afișăm și o valoare agregată la nivel de grup, atunci putem specifica suplimentar în operatorul de selecție:

```
var q = from m in typeof(int).GetMethods()
        group m by m.Name into gb
        select new {Name = gb.Key, Overloads = gb.Count()};
```

Pentru grupare se poate specifica și un comparator (implementare de interfață `IEqualityComparer`) care să spună care este criteriul de determinare a elementelor egale ce formează un grup.

### 11.1.11 Agregare

Există operatorii de agregare: `Count`, `LongCount`, `Sum`, `Min`, `Max`, `Average`, `Aggregate`.

Metoda `Count()` returnează un întreg, iar `LongCount()` un long (întreg pe 64 de biți, cu semn) reprezentând numărul de elemente din secvență.

Metoda `Sum` calculează suma unor valori numerice dintr-o secvență. Metodele sunt:

```
public static Numeric Sum(
    this IEnumerable<Numeric> source);
```

```
public static Numeric Sum<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
```

unde `Numeric` se referă la un tip de date de forma: `int`, `int?`, `long`, `long?`, `double`, `double?`, `decimal`, sau `decimal?`.

Exemplu:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.Sum();
Console.WriteLine(query.ToString());
```



Dacă dorim să determinăm care este suma salariilor anuale primite de către fiecare salariat în parte, atunci putem forma prima oară grupuri bazate pe numele de familie al salariatului (sau și mai bine: după id-ul lui) și apoi se poate face suma salariilor de-a lungul anilor. Presupunem că salariile sunt date astfel:

```
List<Salary> salaries = new List<Salary> {
    new Salary {
        IDPerson = 1,
        Year = 2004,
        SalaryYear = 10000.00 },
    new Salary {
        IDPerson = 1,
        Year = 2005,
        SalaryYear = 15000.00 },
    new Salary {
        IDPerson = 2,
        Year = 2005,
        SalaryYear = 20000.00 }
};
```

```
var query =
    from p in people
    join s in salaries on p.ID equals s.IDPerson
    select new
    {p.FirstName,
     p.LastName,
     s.SalaryYear
    };
```

Apoi:

```
var querySum = from q in query
    group q by q.LastName into gp
    select new {
        LastName = gp.Key,
        TotalSalary = gp.Sum(q => q.SalaryYear)
    };
```

Operatorii Min, Max, Average trebuie apelați numai pentru o colecție de valori numerice.

Operatorul **Aggregate** permite definirea unei metode care să fie folosită pentru sumarizarea datelor. Declarația metodei este:

```

public static T Aggregate<T>(
    this IEnumerable<T> source ,
    Func<T, T, T> func );

public static U Aggregate<T, U>(
    this IEnumerable<T> source ,
    U seed ,
    Func<U, T, U> func );

```

A doua metodă se specifică prin intermediul valorii **seed** care este valoarea de început cu care se pornește agregarea. Dacă nu se specifică nicio valoare pentru **seed**, atunci primul element din colecție este luat drept **seed**.

Pentru înmulțirea valorilor cuprinse într-un tablou se poate proceda astfel:

```

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.Aggregate((a,b) => a * b);

```

Valoarea afișată este  $9! = 362880$ . Specificarea unei valori pentru **seed** se face (exemplu):

```

int[] numbers = { 9, 3, 5, 4, 2, 6, 7, 1, 8 };
var query = numbers.Aggregate(5, (a,b) => ( (a < b) ? (a * b) : a ));

```

Pe măsură ce se iterează peste șirul **numbers** se face acumularea valorii în **seed**.

## 11.2 Exerciții

1. Să se genereze un vector de întregi cu 40 de elemente, toate cu valoarea -1.

```

//varianta naiva
var myNumbers = new int[40];
for (int i = 0; i < myNumbers.Length; i++)
{
    myNumbers[i] = -1;
}

```

Prin LINQ acest lucru se rezolvă cu mai puțin cod:

```

var list = Enumerable.Repeat(-1, 40).ToArray();
//e necesar apelul ToArray din final pentru a transforma
//rezultatul interogarii intr-un tablou

```

2. Să se afișeze elementele unui tablou.

```
//varianta uzuala
foreach(var x in myNumbers)
{
    Console.WriteLine(x.ToString());
}

//varianta LINQ
Array.ForEach(myArray, x => Console.WriteLine(x.ToString()));
```

3. Să se depună într-un tablou numerele de la 1 la 100.

```
//varianta naiva
var myNumbers[] = new int[100];
for(int i=1; i<=100; i++)
{
    myNumbers[i-1] = i;
}

//cu linq
var myNumbers = Enumerable.Range(1, 100).ToArray();
```

4. Să se depună într-o listă primele 20 de litere.

```
var list = Enumerable.Range(0, 20).
    Select(x => (char)('a' + x)).ToList();
```

5. Din lista numerelor de la 2 la 1000 să se determine toate numerele care sunt prime (testarea primalitatii se face prin impartire).

```
var list = Enumerable.Range(2, 999).
    Where(x => x.IsPrime());
//IsPrime este o metoda de extensie

//extensia pentru testarea primalitatii
static class MyExtensionMethods
{
    public static bool IsPrime(this int x)
    {
        //cod...
    }
}
```

6. Din lista de salarii, să se determine care este cel mai mare salariu – ca valoare.

```
var salMax = salaries.Select(s => s.SalaryYear).Max();
```

7. Folosind rezultatul de la punctul precedent, să se determine care sunt salariații care au luat salariul maxim, împreună cu anii în care au luat acea sumă.

```
var list = from p in people
            join s in salaries
            on p.ID equals s.IDPerson
            where s.SalaryYear == salMax
            select new { p.FirstName, p.LastName, s.Year };
```

8. Să se determine salariații care au salariul anual maxim, împreună cu anul în care au luat respectivul salariu și rolul pe care îl au.

```
var list = from p in people
            join s in salaries
            on p.ID equals s.IDPerson
            join r in roles
            on p.IDRole equals r.ID
            where s.SalaryYear == salMax
            select new {
                p.FirstName, p.LastName,
                s.Year, r.RoleDescription };
```

9. Dacă pentru unii salariați se cunosc niste numere de telefon (colecție separată), atunci ce fel de joncțiune realizează “join”? Răspuns: joncțiune interioară, adică se doar date care au corespondență pentru valorile specificate lângă `equals`.

10. Să se creeze o joncțiune la stânga între colecția de salariați și telefoane; pentru salariații care nu au număr de telefon să se scrie “niciun număr” respectiv “nicio descriere” la număr și descriere.

```
var x = from p in people
        from ph in
            phones.Where(ph => ph.PersonID == p.ID).DefaultIfEmpty()
        select new
        {
            p.FirstName,
```

```
p.LastName,
Description = ph == null ?
    "nicio descriere" : ph.Description,
Number = ph == null ?
    "niciun numar" : ph.PhoneNumber
};
```

## Curs 12

# Attribute. Fire de execuție

### 12.1 Attribute

O aplicație .NET conține cod, date și *metadata*. Metadata este o dată despre assembly (versiune, producator etc.), tipuri de date conținute etc stocate împreună cu codul compilat.

Atributele sunt folosite pentru a da o extra-informație compilatorului de .NET. Java folosește adnotări în acest scop. Aceste comentarii însă nu vor fi incluse în bytcode-ul final, doar în eventuala documentație. Folosind atributele, în .NET Framework această informație poate fi stocată în codul compilat și reutilizată ulterior la rulare.

Unde ar fi utile aceste atribute? Un exemplu de utilizare a lor ar fi urmărirea bug-urilor care există într-un sistem sau urmărirea stadiului proiectului. De asemenea, anumite atribute predefinite sunt utilizate pentru a specifica dacă un tip este sau nu serializabil, care sunt porțiunile de cod scoase din circulație, informații despre versiunea assembly-ului etc.

#### 12.1.1 Generalități

Atributele sunt de două feluri: intrinseci (predefinite) și definite de utilizator. Cele intrinseci sunt integrate în platforma .NET. Atributele definite de utilizator sunt create în funcție de dorințele acestuia.

Atributele se pot specifica pentru: assembly-uri, clase, constructori, delegați, enumerări, evenimente, câmpuri, interfețe, metode, module, parametri, proprietăți, indexatori, valori de retur, structuri.

Ținta unui atribut specifică cui anume i se va aplica acel atribut. Tabelul [12.1](#) conține țintele posibile și o scurtă descriere a lor.

Tabelul 12.1: Țintele atributelor.

Țintă	Descriere
All	Orice element
Assembly	Un assembly
ClassMembers	Orice membru al unei clase
Class	O clasă
Constructor	Un constructor
Delegate	Un delegat
Enum	O enumerare
Event	Un eveniment
Field	Un câmp
Interface	O interfață
Method	O metodă
Module	Un modul
Parameter	Un parametru
Property	O proprietate
ReturnValue	O valoare de retur
Struct	O structură

Aplicarea atributelor se face prin specificarea numelui lor între paranteze drepte; mai multe atribute fie se specifică unul deasupra celuilalt, fie în interiorul acelorași paranteze, despărțite prin virgulă. Exemplu:

```
[Serializable]
[WebService]
```

echivalent cu:

```
[Serializable, WebService]
```

În cazul atributelor ce se specifică pentru un assembly, forma lor este:

```
[assembly:AssemblyDelaySign(false)]
```

în cazul acestor din urmă atribute, specificarea lor se face după toate declarațiile `using`, dar înaintea începerii scrierii codului propriu-zis. În general, specificarea unui atribut se face prin scrierea lui imediat înaintea elementului asupra căruia se aplică:

```
using System;
[Serializable]
```

```
class ClasaSerializabila
{
//definitia clasei
}
```

### 12.1.2 Atribute predefinite

Tabelul 12.2 prezintă câteva dintre ele:

Tabelul 12.2: Atribute predefinite.

Atribut	Descriere
System.SerializableAttribute [Serializable]	Permite unei clase să fie serializată pe disc sau într-o rețea
System.NonSerializedAttribute [NonSerialized]	Permite unor membri să nu fie salvați prin serializare
System.Web.Services.WebServiceAttribute [WebService]	Permite specificarea unui nume și a unei descrieri pentru un serviciu Web
System.Web.Services.WebMethodAttribute [WebMethod]	Marchează o metodă ca fiind expusă ca parte a unui serviciu Web
System.AttributeUsageAttribute [AttributeUsage]	Definește parametrii de utilizare pentru atribute
System.ObsoleteAttribute [Obsolete]	Marchează o secvență ca fiind scoasă din uz
System.Reflection.AssemblyVersionAttribute [AssemblyVersion]	Specifică numărul de versiune al unui assembly
System.Attribute.CLSCompliant [CLSCompliant]	Indică dacă un element de program este compatibil cu CLS
System.Runtime.InteropServices.DllImportAttribute [DllImport]	Specifică fișierul dll care conține implementarea pentru o metodă externă

Între parantezele drepte se arată cum se specifică acel atribut.

Exemplul de mai jos folosește atributul *System.ObsoleteAttribute*.

```
using System;
```

```
namespace AttributeSample1
{
    class Class1
    {
        static void Main(string[] args)
```



```

    {
        int s1 = AddTwoNumbers(2, 3);
        int s2 = AddNumbers(2, 3);
        int s3 = AddNumbers(2, 3, 4);
    }

    [Obsolete("obsolete: use AddNumbers instead")]
    public static int AddTwoNumbers(int a, int b)
    {
        return a + b;
    }

    public static int AddNumbers(params int[] numbers)
    {
        return numbers.Sum();
    }
}

```

La compilarea codului se generează un avertisment care semnalizează faptul că se utilizează o metodă care este scoasă din uz. Mesajul specificat ca parametru al atributului este afișat ca mesaj al avertismentului. Suplimentar, pentru atributul `Obsolete` se poate specifica dacă respectivul avertisment este sau nu interpretat ca o eroare, adăugând un parametru de tip `bool`: *false* ca la compilare să se genereze avertisment, *true* pentru ca utilizarea să fie tratată ca o eroare.

Exemplul de mai jos exemplifică serializarea și deserializarea unui obiect:

```

using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Point2D
{
    public int X{get; set;}
    public int Y{get; set;}
}
class MyMainClass
{
    public static void Main()
    {
        Point2D my2DPoint = new Point2D();
    }
}

```

```

    my2DPoint.X = 100;
    my2DPoint.Y = 200;
    Stream writeStream = File.Create("Point2D.bin");
    BinaryFormatter binaryWrite = new BinaryFormatter();
    binaryWrite.Serialize(writeStream, my2DPoint);
    writeStream.Close();
    Point2D aNewPoint = new Point2D();
    Console.WriteLine("New Point Before Deserialization: ({0}, {1})",
        aNewPoint.X.ToString(), aNewPoint.Y.ToString());
    Stream readStream = File.OpenRead("Point2D.bin");
    BinaryFormatter binaryRead = new BinaryFormatter();
    aNewPoint = binaryRead.Deserialize(readStream) as Point2D;
    readStream.Close();
    Console.WriteLine("New Point After Deserialization: ({0}, {1})",
        aNewPoint.X.ToString(), aNewPoint.Y.ToString());
}
}

```

### 12.1.3 Exemplificarea altor atribute predefinite

#### Atributul *Conditional*

Acest atribut se atașează la o metodă pentru care se dorește ca atunci când compilatorul o întâlnește la un apel, dacă un anumit simbol nu e definit atunci nu va fi chemată. Este folosită pentru a omite anumite apeluri de metode (de exemplu cele folosite la etapa de debugging).

Exemplu:

```

using System;
using System.Diagnostics;

namespace CondAttrib
{
    class Thing
    {
        private string name;
        public Thing(string name)
        {
            this.name = name;
            SomeDebugFunc();
            SomeFunc();
        }
    }
}

```

```
public void SomeFunc()
{ Console.WriteLine("SomeFunc"); }

[Conditional("DEBUG")]
public void SomeDebugFunc()
{ Console.WriteLine("SomeDebugFunc"); }
}

public class Class1
{
    static void Main(string[] args)
    {
        Thing t = new Thing("T1");
    }
}
```

Definirea unui anumit simbol (de exemplu DEBUG) se poate face în două moduri:

- prin folosirea unei directive de preprocesor de tipul `#define` în fișier `.cs`:

```
#define DEBUG
```

- prin folosirea parametrilor din linia de comanda sau a posibilităților mediului integrat de dezvoltare. De exemplu, pentru a se defini un anumit simbolul din Visual Studio se procedează astfel: clic dreapta în Solution Explorer pe numele proiectului, selectare Properties apoi Configuration Properties. Pe linia Conditional Compilation Constants se adaugă simbolul dorit. Acest lucru va avea ca efect folosirea opțiunii *define* a compilatorului.

Dacă la compilare simbolul nu este definit atunci compilatorul va ignora apelurile de metode calificate cu atributul Conditional. Acest lucru se poate verifica atât urmărind execuția programului, cât și din inspecția codului CIL rezultat la compilare.

### Atributul CLSCompliant

Acest atribut se aplică pe assembly-uri. Dacă un assembly este marcat ca fiind *CLSCompliant*, orice tip expus public în assembly care nu este compatibil CLS trebuie să fie marcat cu *CLSCompliant(false)*. De exemplu, CLS

prevede faptul că tipurile de date întregi ar trebui să fie cu semn. O clasă poate să conțină membri (câmpuri, metode) de tip unsigned, dar dacă respectiva clasă este declarată ca fiind CLSCompliant atunci acestea ar trebui declarate ca fiind ne-vizibile din afara assembly-ului. Dacă ele sunt expuse în afara assembly-ului, unele limbaje .NET s-ar putea să nu le poată folosi. Pentru a ajuta dezvoltatorii .NET să evite asemenea situații, platforma pune la dispoziție atributul CLSCompliant cu care se controlează răspunsul compilatorului la expunerea entităților ne-compatibile cu CLS; astfel, se va genera o eroare sau se vor ignora asemenea cazuri.

Exemplu:

```
[assembly:CLSCompliant(true)]
namespace DemoCLS
{
    public class ComplianceTest
    {
        //Tipul uint nu este compatibil CLS
        //deoarece acest camp este privat, regulile CLS nu se aplica
        private uint a = 4;

        //deoarece acest camp uint este public, avem
        //incompatibilitate CLS
        public uint B = 5;

        //Acesta este modul corect de expunere a unui uint:
        //tipul long este compatibil CLS
        public long A
        {
            get { return a; }
        }
    }
}
```

Dacă se compilează assembly-ul de mai sus atunci apare o eroare:

```
Type of 'DemoCLS.ComplianceTest.B' is not CLS-compliant
```

Pentru ca această eroare să nu mai apară putem să declarăm B ca fiind invizibil din exteriorul assembly-ului sau să adăugăm înaintea lui B atributul:

```
[CLSCompliant(false)]
```

Meritul acestui atribut este că anunță programatorul despre eventualele neconcordanțe cu Common Language Specifications.

### 12.1.4 Atribute definite de utilizator

În general, atributele predefinite acoperă marea majoritate a situațiilor care cer utilizarea de atribute. Eventualizarea ca utilizatorul să dorească crearea propriilor sale atribute este prevăzută de către platforma .NET, dându-se posibilitatea definirii lor.

Există câteva situații în care e benefică definirea de noi atribute. Exemplul cel mai des întâlnit este acela în care pentru a se menține informații despre un cod la care lucrează mai multe echipe, se definește un atribut care să servească la pasarea de informație relativ la porțiuni din cod. Un alt exemplu este utilizarea unui sistem de urmărire a stadiului de dezvoltare a codului, care ar folosi informația stocată în atribute.

Un atribut utilizator este o clasă. Se cere a fi derivată din *System.Attribute* fie direct, fie indirect. Sunt câțiva pași care trebuie parcurși pentru realizarea unui atribut: specificarea țintei, derivarea adecvată a clasei atribut, denumirea clasei în conformitate cu anumite reguli (recomandat, dar nu obligatoriu) și definirea clasei.

#### Pasul 1. Declararea țintei

Primul pas în crearea unui atribut utilizator este specificarea domeniului său de aplicabilitate, adică a elementelor cărora li se poate atașa. Țintele sunt cele din tabelul 12.1, care sunt de fapt valori din enumerarea *AttributeTargets*. Specificarea țintei se face prin intermediul unui (meta)atribut *AttributeUsage*. Pe lângă ținta propriu-zisă, se mai pot specifica valorile proprietăților *Inherited* și *AllowMultiple*.

Proprietatea *Inherited* este de tip boolean și precizează dacă atributul poate fi moștenit de către clasele derivate din cele căreia i se aplică. Valoarea implicită este *true*.

Proprietatea *AllowMultiple* este de asemenea de tip boolean și specifică dacă se pot utiliza mai multe instanțe ale atributului pe un același element. Valoarea implicită este *false*.

Vom defini mai jos ținta atributului pe care îl vom construi ca fiind *assembly-ul*, *clasa*, *metoda*, cu posibilitate de repetare a sa:

```
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
    | AttributeTargets.Method, AllowMultiple=true)]
```

#### Pasul 2. Declararea unei clase atribut

Pentru declararea unei clase atribut, următoarele reguli trebuie să fie avute în vedere:

- (obligatoriu) O clasă atribut trebuie să deriveze direct sau indirect *System.Attribute*
- (obligatoriu) O clasă atribut trebuie să fie declarată ca fiind publică
- (recomandat) Denumirea unui atribut ar trebui să aibă sufixul *Attribute*.

Vom declara clasa atribut *CodeTrackerAttribute*. Acest atribut s-ar putea folosi în cazul în care s-ar dori urmărirea dezvoltării codului în cadrul unui proiect de dimensiuni foarte mari, la care participă mai multe echipe. Acest atribut utilizator va fi inclus în codul compilat și distribuit altor echipe (care nu vor avea acces la cod). O alternativă ar fi folosirea documentației XML generate după comentariile din cod.

```
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
                | AttributeTargets.Method, AllowMultiple=true)]
public class CodeTrackerAttribute : System.Attribute
{
    //cod
}
```

### Pasul 3. Declararea constructorilor și a proprietăților

Acest pas constă în definirea efectivă membrilor clasei. Vom considera că atributul pe care îl creăm va purta trei informații: numele programatorului care a acționat asupra respectivei unități de cod, faza în care se află și note opționale. Primele două atribute sunt mandatorii și vor fi preluate prin constructor, al treilea poate fi setat prin intermediul unei proprietăți.

```
using System;
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
                | AttributeTargets.Method, AllowMultiple=true)]
public class CodeTrackerAttribute : System.Attribute
{
    private string name;
    private string phase;
    private string notes;

    public CodeTrackerAttribute(string name, string phase)
    {
        this.name = name;
        this.phase = phase;
    }
}
```

```
    }

    public virtual string Name
    {
        get{return name;}
    }

    public virtual string Phase
    {
        get{return phase;}
    }

    public virtual string Notes
    {
        get{return notes;}
        set{notes=value;}
    }
}
```

#### **Pasul 4. Utilizarea atributului utilizator**

Atributele definite de utilizator sunt folosite în același mod ca și cele implicite. Codul de mai jos exemplifică acest lucru:

```
[CodeTracker("Lucian Sasu", "implementing specification",
    Notes = "First attempt")]
class AttribTest
{
    public AttribTest()
    {
        Console.WriteLine("AttribTest instance");
    }
    [CodeTracker("Lucian Sasu", "May 25th 2011")]
    public void SayHello(String message)
    {
        Console.WriteLine(message);
    }
}
```

O inspecție a codului CIL rezultat arată că aceste atribute s-au salvat în codul compilat. Obținerea acestor atribute și a valorilor lor se poate face prin reflectare.

## 12.2 Fire de execuție

Firele de execuție<sup>1</sup> sunt responsabile cu multitasking-ul în interiorul unei aplicații. Clasele și interfețele responsabile pentru crearea aplicațiilor multifer se găsesc în spațiul de nume *System.Threading*. Vom discuta în cele ce urmează despre managementul thread-urilor și despre sincronizare.

De cele mai multe ori crearea de thread-uri este necesară pentru a da utilizatorului impresia că programul execută mai multe acțiuni simultan, în cadrul unei aplicații. Pentru ca o interfață utilizator să poată fi folosită fără a se aștepta încheierea unei anumite secvențe de instrucțiuni, e nevoie de mai multe fire de execuție (unele pentru procesarea efectivă a informației, altele care să răspundă acțiunilor utilizatorului). Pentru probleme de calcul numeric, exprimarea procesului de calcul se poate face foarte natural sub formă de fire de execuție. De asemenea, strategia de rezolvare “divide et impera” se pretează natural la o lucrul cu fire de execuție. În sfârșit, se poate beneficia de sisteme cu procesoare hyperthreading, multiprocesor, multicore sau combinații ale acestora.

## 12.3 Managementul thread-urilor

### 12.3.1 Pornirea thread-urilor

Cea mai simplă metodă de a crea un fir de execuție este de a crea o instanță a clasei *Thread*, al cărei constructor preia un singur argument de tip delegat. În BCL este definit tipul delegat *ThreadStart*, care este folosit ca prototip pentru orice metodă care se vrea a fi lansată într-un fir de execuție. Declarația de *ThreadStart* este:

```
public delegate void ThreadStart();
```

Crearea unui fir de execuție se face pe baza unei metode care returnează *void* și nu preia nici un parametru:

```
ThreadStart ts = new ThreadStart(myFunc);  
Thread myThread = new Thread( ts );
```

Notă: prima linie se poate scrie mai pe scurt:

```
ThreadStart ts = myFunc;
```

Un exemplu simplu este:

---

<sup>1</sup>Engl: threads; vom folosi alternativ termenii “thread” și “fir de execuție”.



```

using System;
using System.Threading;

class SimpleThreadApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("[WorkerThreadMethod] Worker " +
            "thread started");
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);
        Console.WriteLine("[Main] Creating worker thread");
        Thread t = new Thread(worker);
        t.Start();
        Console.WriteLine( "[Main] Have requested the start of
            worker thread");
        Console.ReadLine();
    }
}

```

Până la linia `t.Start()` există un singur thread: cel dat de pornirea metodei `Main`. După `t.Start()` sunt 2 thread-uri: cel anterior și `t`. Primul mesaj din `Main` va fi tipărit înaintea celui din thread-ul `t`; în funcție de cum anume se planifică thread-urile pentru execuție, mesajul de după `t.Start()` poate să apară înainte sau după mesajul tipărit de metoda `WorkerThreadMethod()`. De remarcat că simpla creere a firului de execuție nu determină și pornirea lui: acest lucru se întâmplă după apelul metodei `Start` definită în clasa `Thread`.

Exemplul următor pornește două fire de execuție. Funcțiile care conțin codul ce se va executa în câte un thread sunt `Increment()` și `Decrement()`:

```

using System;
using System.Threading;
class Tester
{
    static void Main( )
    {
        Tester t = new Tester( );
        t.DoTest( );
    }
}

```

```
}  
public void DoTest( )  
{  
    // creeaza un thread pentru Incrementer  
    Thread t1 = new Thread( new ThreadStart(Incrementer) );  
    // creeaza un thread pentru Decrementer  
    Thread t2 = new Thread( new ThreadStart(Decrementer) );  
    // porneste threadurile  
    t1.Start( );  
    t2.Start( );  
}  
  
public void Incrementer( )  
{  
    for (int i = 1;i<=1000;i++)  
    {  
        Console.WriteLine( "Incrementer: {0}", i);  
    }  
}  
  
public void Decrementer( )  
{  
    for (int i = 1000;i>0;i--)  
    {  
        Console.WriteLine("Decrementer: {0}", i);  
    }  
}  
}
```

La ieșire se vor mixa mesajele tipărite de primul thread cu cele tipărite de cel de-al doilea thread:

```
...  
Incrementer: 102  
Incrementer: 103  
Incrementer: 104  
Incrementer: 105  
Incrementer: 106  
Decrementer: 1000  
Decrementer: 999  
Decrementer: 998  
Decrementer: 997
```

...

Perioada de timp alocată fiecărui thread este determinată de către planificatorul de fire de execuție<sup>2</sup> și depinde de factori precum viteza procesorului, gradul lui de ocupare etc.

O altă modalitate de obținere a unei referințe la un thread este prin apelul proprietății statice `Thread.CurrentThread` pentru firul de execuție curent. În exemplul de mai jos se obține obiectul `Thread` asociat firului de execuție curent, i se setează numele (proprietate read/write de tip `String`) și se afișează pe ecran:

```
Thread current = Thread.CurrentThread;
current.Name = "My name";
Console.WriteLine("nume={0}", current.Name );
```

### 12.3.2 Metoda *Join()*

Există situații în care, înaintea unei instrucțiuni trebuie să se asigure faptul că un alt fir de execuție *t* s-a terminat; acest lucru se va face folosind apelul *t.Join()* înaintea instrucțiunii în cauză; în acest moment firul de execuție care a apelat *t.Join()* intră în așteptare (e suspendat).

Dacă de exemplu în metoda *Main* se lansează o colecție de thread-uri (stocată în *myThreads*), atunci pentru a se continua execuția numai după ce toate firele din colecție s-au terminat, se procedează astfel:

```
foreach( Thread myThread in myThreads )
{
    myThread.Join();
}
Console.WriteLine("All my threads are done");
```

Mesajul final se va tipări doar când toate firele de execuție s-au terminat.

### 12.3.3 Suspendarea firelor de execuție

Se poate ca în anumite cazuri să se dorească suspendarea unui fir de execuție pentru o scurtă perioadă de timp. Clasa *Thread* oferă o metodă statică supraîncărcată *Sleep()*, care poate prelua un parametru de tip *int* reprezentând milisecundele de "adormire", iar a doua variantă preia un argument de tip *TimeSpan*, care reprezintă cea mai mică unitate de timp

---

<sup>2</sup>Engl: thread scheduler.

care poate fi specificată, egală cu 100 nanosecunde. Pentru a cere firului de execuție curent să se suspende pentru o secundă, se execută în cadrul acestuia:

```
Thread.Sleep(1000);
```

În acest fel se semnalează planificatorului de fire de execuție că poate lansa un alt thread.

Dacă în exemplul de mai sus se adaugă un apel *Thread.Sleep(1)* după fiecare *WriteLine()*, atunci ieșirea se schimbă dramatic:

```
Iesire (extras)
Incrementer: 0
Incrementer: 1
Decrementer: 1000
Incrementer: 2
Decrementer: 999
Incrementer: 3
Decrementer: 998
Incrementer: 4
Decrementer: 997
Incrementer: 5
Decrementer: 996
Incrementer: 6
Decrementer: 995
```

### 12.3.4 Omorârea thread-urilor

De obicei, un fir de execuție moare după ce se termină de executat. Se poate totuși cere unui fir de execuție să își înceteze execuția folosind metoda *Abort()*. Acest lucru va duce la aruncarea unei excepții în interiorul firului de execuție căruia i se cere suspendarea: *Thread.AbortedException*, pe care firul respectiv o poate prinde și trata, permițându-i eliberarea de resurse alocate. Ca atare, se recomandă ca o metodă care se va lansa ca fir de execuție să se compună dintr-un bloc *try* care conține instrucțiunile utile, după care un bloc *catch* și/sau *finally* care vor efectua eliberarea de resurse.

Exemplu:

```
using System;
using System.Threading;
class Tester
{
```

```
static void Main( )
{
    Tester t = new Tester( );
    t.DoTest( );
}
public void DoTest( )
{
    // creeaza un vector de threaduri
    Thread[] myThreads =
    {
        new Thread( new ThreadStart(Decrementer) ),
        new Thread( new ThreadStart(Incrementer) ),
        new Thread( new ThreadStart(Incrementer) )
    };
    // porneste fiecare thread
    int ctr = 1;
    foreach (Thread myThread in myThreads)
    {
        myThread.IsBackground=true;
        myThread.Start( );
        myThread.Name = "Thread" + ctr.ToString( );
        ctr++;
        Console.WriteLine("Started thread {0}", myThread.Name);
        Thread.Sleep(50);
    }
    // dupa ce firele se pornesc,
    // comanda oprirea threadului 1
    myThreads[1].Abort( );
    // asteapta ca fiecare thread sa se termine
    foreach (Thread myThread in myThreads)
    {
        myThread.Join( );
    }
    Console.WriteLine("All my threads are done.");
}
// numara descrescator de la 1000
public void Decrementer( )
{
    try
    {
        for (int i = 1000;i>0;i--)
```

```
        {
            Console.WriteLine("Thread {0}. Decrementer: {1}",
                Thread.CurrentThread.Name, i);
            Thread.Sleep(1);
        }
    }
    catch (ThreadAbortedException)
    {
        Console.WriteLine(
            "Thread {0} interrupted! Cleaning up...",
            Thread.CurrentThread.Name);
    }
    finally
    {
        Console.WriteLine("Thread {0} Exiting. ",
            Thread.CurrentThread.Name);
    }
}
// numara cresacator pana la 1000
public void Incrementer( )
{
    try
    {
        for (int i =1;i<=1000;i++)
        {
            Console.WriteLine("Thread {0}. Incrementer: {1}",
                Thread.CurrentThread.Name, i);
            Thread.Sleep(1);
        }
    }
    catch (ThreadAbortedException)
    {
        Console.WriteLine( "Thread {0} interrupted! Cleaning up...",
            Thread.CurrentThread.Name);
    }
    finally
    {
        Console.WriteLine( "Thread {0} Exiting. ",
            Thread.CurrentThread.Name);
    }
}
```

```
}
```

Ieșire:

```
Started thread Thread1
Thread Thread1. Decrementer: 1000
Thread Thread1. Decrementer: 999
Thread Thread1. Decrementer: 998
Started thread Thread2
Thread Thread1. Decrementer: 997
Thread Thread2. Incrementer: 0
Thread Thread1. Decrementer: 996
Thread Thread2. Incrementer: 1
Thread Thread1. Decrementer: 995
Thread Thread2. Incrementer: 2
Thread Thread1. Decrementer: 994
Thread Thread2. Incrementer: 3
Started thread Thread3
Thread Thread1. Decrementer: 993
Thread Thread2. Incrementer: 4
Thread Thread2. Incrementer: 5
Thread Thread1. Decrementer: 992
Thread Thread2. Incrementer: 6
Thread Thread1. Decrementer: 991
Thread Thread3. Incrementer: 0
Thread Thread2. Incrementer: 7
Thread Thread1. Decrementer: 990
Thread Thread3. Incrementer: 1
Thread Thread2 interrupted! Cleaning up...
Thread Thread2 Exiting.
Thread Thread1. Decrementer: 989
Thread Thread3. Incrementer: 2
Thread Thread1. Decrementer: 988
Thread Thread3. Incrementer: 3
Thread Thread1. Decrementer: 987
Thread Thread3. Incrementer: 4
Thread Thread1. Decrementer: 986
Thread Thread3. Incrementer: 5
// ...
Thread Thread1. Decrementer: 1
Thread Thread3. Incrementer: 997
```

### 12.3.5 Sugerarea priorităților firelor de execuție

Un fir de execuție se lansează implicit cu prioritatea *ThreadPriorityLevel.Normal*. Dar scheduler-ul poate fi influențat în activitatea sa prin setarea de diferite nivele de prioritate pentru fire; aceste nivele fac parte din enumerarea *ThreadPriorityLevel*: *ThreadPriorityLevel.TimeCritical*, *ThreadPriorityLevel.Highest*, *ThreadPriorityLevel.AboveNormal*, *ThreadPriorityLevel.Normal*, *ThreadPriorityLevel.BelowNormal*, *ThreadPriorityLevel.Lowest*, *ThreadPriorityLevel.Idle*. Prioritatea este descrescătoare în lista prezentată. Pe baza priorității procesului care conține firele de execuție și a priorității firelor, se calculează un nivel de prioritate (de ex. pe mașini Intel valori între 0 și 31) care determină prioritatea în ansamblul sistemului de operare a firului respectiv.

Setarea unei anumite priorități se face folosind proprietatea *Priority*:

```
myThread.Priority = ThreadPriorityLevel.Highest;
```

### 12.3.6 Fire în fundal și fire în prim-plan

Relativ la proprietatea booleană *IsBackground*, trebuie făcută precizarea că un fir de execuție poate să se execute în fundal (background) sau în prim plan (foreground). Diferența dintre cele două posibilități o constituie faptul că dacă un proces are măcar un fir de execuție în foreground, CLR va menține aplicația în execuție. O dată ce toate firele de execuție de tip foreground se termină, CLR va executa *Abort()* pentru fiecare fir de execuție de tip background (dacă mai există așa ceva) și termină procesul.

Exemplu:

```
using System;
using System.Threading;

class Test
{
    static void Main()
    {
        BackgroundTest shortTest = new BackgroundTest(10);
        Thread foregroundThread =
            new Thread(new ThreadStart(shortTest.RunLoop));
        foregroundThread.Name = "ForegroundThread";

        BackgroundTest longTest = new BackgroundTest(50);
        Thread backgroundThread =
            new Thread(new ThreadStart(longTest.RunLoop));
```



```
        backgroundThread.Name = "BackgroundThread";
        backgroundThread.IsBackground = true;

        foregroundThread.Start();
        backgroundThread.Start();
    }
}

class BackgroundTest
{
    int maxIterations;

    public BackgroundTest(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public void RunLoop()
    {
        String threadName = Thread.CurrentThread.Name;

        for(int i = 0; i < maxIterations; i++)
        {
            Console.WriteLine("{0} count: {1}",
                               threadName, i.ToString());
            Thread.Sleep(250);
        }
        Console.WriteLine("{0} finished counting.", threadName);
    }
}
```

Firul din foreground va menține procesul în execuție până când se termină ciclul său `while`. Când acesta se termină, procesul este oprit, chiar dacă ciclul `while` din firul de execuție din background nu și-a terminat execuția.

## 12.4 Sincronizarea

Sincronizarea se ocupă cu controlarea accesului la resurse partajate de mai multe fire de execuție. De exemplu, se poate cere ca utilizarea unei resurse anume să se facă la un moment dat de către un singur fir de execuție. Vom discuta aici trei mecanisme de sincronizare: clasa *Interlock*, instrucțiunea

C# *lock* și clasa *Monitor*. Drept resursă partajată vom considera un câmp întreg.

```
public void Incrementer( )
{
    try
    {
        while (counter < 1000)
        {
            int temp = counter;
            temp++; // increment
            // simuleza o sarcina oarecare in acest thread
            Thread.Sleep(1);
            // atribuie valoarea incrementata
            // variabilei counter
            // si afiseaza rezultatul
            counter = temp;
            Console.WriteLine( "Thread {0}. Incrementer: {1}",
                Thread.CurrentThread.Name, counter);
        }
    }
    catch (ThreadAbortedException)
    {
        Console.WriteLine( "Thread {0} interrupted! Cleaning up...",
            Thread.CurrentThread.Name);
    }
    finally
    {
        Console.WriteLine( "Thread {0} Exiting. ",
            Thread.CurrentThread.Name);
    }
}
```

Câmpul *counter* se inițializează cu 0. Să presupunem că pornim două fire de execuție pe baza metodei *Incrementer()* de mai sus. Este posibil să se întâmple următoarele: primul thread va citi valoarea lui *counter* (0) și o va atribui variabilei temporare *temp*, pe care o va incrementa apoi. Al doilea fir se activează și el, va citi valoarea nemodificată a lui *counter* și va atribui această valoare lui *temp*. Primul fir de execuție își termină munca, apoi asignează valoarea variabilei temporare (1) lui *counter* și o afișează. Al doilea thread face exact același lucru. Se tipărește astfel 1, 1. La următoarele

iterații se va afișa 2, 2, 3, 3 etc, în locul intenționatei secvențe 1, 2, 3, 4. Mai pot exista și alte scenarii nefavorabile.

Exemplu:

```
using System;
using System.Threading;
class Tester
{
    private int counter = 0;
    static void Main( )
    {
        Tester t = new Tester( );
        t.DoTest( );
    }
    public void DoTest( )
    {
        Thread t1 = new Thread( new ThreadStart(this.Incrementer) );
        t1.IsBackground=true;
        t1.Name = "ThreadOne";
        t1.Start( );
        Console.WriteLine("Started thread {0}", t1.Name);
        Thread t2 = new Thread( new ThreadStart(this.Incrementer) );
        t2.IsBackground=true;
        t2.Name = "ThreadTwo";
        t2.Start( );
        Console.WriteLine("Started thread {0}", t2.Name);
        t1.Join( );
        t2.Join( );
    }
    // numara crescator pana la 1000
    public void Incrementer( )
    {
        //la fel ca la inceputul sectiunii
    }
}
```

Ieșire:

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadOne. Incrementer: 2
```

```

Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
Thread ThreadOne. Incrementer: 4
Thread ThreadTwo. Incrementer: 5
Thread ThreadOne. Incrementer: 5
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 6

```

Trebuie deci să se realizeze o excludere reciprocă a thread-urilor pentru accesul la *counter*.

### 12.4.1 Clasa *Interlocked*

Incrementarea și decrementarea unei valori este o situație atât de des întâlnită, încât C# pune la dispoziție o clasă specială *Interlocked* pentru o rezolvare rapidă. Clasa include două metode statice, *Increment()* și *Decrement()*, care incrementează sau decrementează o valoare, însă sub un control sincronizat.

Putem modifica metoda *Incrementer()* de mai sus astfel:

```

public void Incrementer( )
{
    try
    {
        while (counter < 1000)
        {
            Interlocked.Increment(ref counter);
            // simuleaza o sarcina in aceasta metoda
            Thread.Sleep(1);
            // asigura valoarea decrementata
            // si afiseaza rezultatul
            Console.WriteLine(
                "Thread {0}. Incrementer: {1}",
                Thread.CurrentThread.Name,
                counter);
        }
    }
    //blocurile catch si finally raman neschimbate
}

```

Ieșirea este cea dorită:

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadTwo. Incrementer: 2
Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
Thread ThreadOne. Incrementer: 5
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 7
Thread ThreadTwo. Incrementer: 8
```

### 12.4.2 Instrucțiunea *lock*

Există situații când vrem să blocăm alte variabile decât cele de tip `int`. Un `lock` marchează o secțiune critică a codului, producând astfel sincronizare pentru un obiect. La utilizare, se specifică un obiect pentru care se stabilește un `lock`, după care o instrucțiune sau un grup de instrucțiuni. `Lock`-ul este înlăturat la sfârșitul instrucțiunii/blocului de instrucțiuni. Sintaxa este:

```
lock(expresie)
{
    instructiuni
}
```

Exemplu: metoda *Incrementer()* se va modifica după cum urmează:

```
public void Incrementer( )
{
    try
    {
        while (counter < 1000)
        {
            lock (this)
            {
                int temp = counter;
                temp++;
                Thread.Sleep(1);
                counter = temp;
            }
            // atribuie valoarea decrementata
            // si afiseaza rezultatul
            Console.WriteLine( "Thread {0}. Incrementer: {1}",
```

```

        Thread.CurrentThread.Name, counter);
    }
}
//blocurile catch si finally raman neschimbate
}

```

Rezultatele sunt afișate exact ca la secțiunea 12.4.1.

### 12.4.3 Clasa *Monitor*

Clasa *Monitor* conține metode pentru a controla sincronizarea firelor de execuție, permițând declararea unei zone critice în care la un moment dat doar un thread trebuie să opereze.

Atunci când se dorește să se înceapă sincronizarea, se va apela metoda *Enter*, dând obiectul pentru care se va face blocarea:

```
Monitor.Enter( obiect );
```

Dacă monitorul este nedisponibil, atunci înseamnă că un alt thread este într-o regiune critică a obiectului respectiv. Se mai poate folosi de asemenea metoda *Wait()*, care eliberează monitorul, dar blochează threadul, informând CLR că atunci când monitorul devine din nou liber, thread-ul curent ar vrea să își continue execuția (este adăugat într-o coadă de așteptare formată din fire de execuție blocate pe obiect). Terminarea zonei critice se face folosind metoda *Exit()* a clasei *Monitor*. Metoda *Pulse()* semnalează că a avut loc o schimbare de stare, în urma căreia este posibil ca un alt fir de execuție care așteaptă va putea să fie continuat (ordinea de selectare a firelor ce vor fi executate fiind ordinea introducerii în coada de așteptare). Înrudită este metoda *PulseAll()* care anunță toate obiectele blocate pe un anumit obiect de schimbarea de stare.

Să presupunem că avem o clasă *MessageBoard* unde fire individuale pot citi și scrie mesaje. Vom sincroniza accesul la această clasă astfel încât doar un thread să poată acționa la un moment dat. Clasa *MessageBoard* va avea o metodă *Reader()* și una *Writer()*.

Metoda *Reader()* determină dacă string-ul *message* conține vreun mesaj valabil, iar metoda *Writer()* va scrie în acest string. Dacă nu sunt mesaje în timpul citirii, thread-ul *Reader()* va intra în stare de așteptare folosind *Wait()* până când metoda *Writer()* scrie un mesaj și transmite un mesaj via *Pulse()* pentru a trezi alte thread-uri.

```

using System;
using System.Threading;

```

```
class MessageBoard
{
    private String messages = "no messages" ;
    public void Reader()
    {
        try
        {
            Monitor.Enter(this);
            //daca nu e nici un mesaj atunci asteapta
            if (messages == "no messages")
            {
                Console.WriteLine("{0} {1}",
                    Thread.CurrentThread.Name, messages);
                Console.WriteLine("{0} waiting...",
                    Thread.CurrentThread.Name);
                Monitor.Wait(this);
            }
            //inseamna ca mesajul s-a schimbat
            Console.WriteLine("{0} {1}",
                Thread.CurrentThread.Name, messages);
        }
        finally
        {
            Monitor.Exit(this);
        }
    }
    public void Writer()
    {
        try
        {
            Monitor.Enter(this);
            messages = "Greetings Caroline and Marianne!";
            Console.WriteLine("{0} Done writing message...",
                Thread.CurrentThread.Name);
            //semnaleaza threadului de asteptare ca s-a schimbat mesajul
            Monitor.Pulse(this);
        }
        finally
        {
            Monitor.Exit(this);
        }
    }
}
```

```

    }
    public static void Main()
    {
        MessageBoard myMessageBoard = new MessageBoard();
        Thread reader = new Thread(new
        ThreadStart(myMessageBoard.Reader));
        reader.Name = "ReaderThread:";
        Thread writer = new Thread( new
        ThreadStart(myMessageBoard.Writer));
        writer.Name = "WriterThread:";
        reader.Start();
        writer.Start();
    }
}

```

Ieșirea este:

```

ReadrThread: no messages
ReaderThread: waiting...
WriterThread: Done writing message...
ReaderThread: Greetings Caroline and Marianne!

```

După cum se vede mai sus, thread-ul *reader* este pornit primul, el blochează clasa obiectul de tip *MessageBoard*, ceea ce înseamnă că are acces exclusiv la variabila *message*. Dar deoarece *message* nu conține nici un mesaj, thread-ul *reader* eliberează obiectul pe care l-a blocat mai înainte prin apelul *Wait()*. Thread-ul *writer* va putea acum să blocheze obiectul pentru a scrie mesajul. Apoi el cheamă metoda *Pulse()* pentru a semnala firului *reader* că a apărut o schimbare de stare a obiectului indicat wde *this*.



# Curs 13

## Noutăți în C# 4.0

### 13.1 Parallel LINQ

Parallel LINQ (PLINQ) permite executarea de interogări în mod paralel, pentru stituațiile în care există un sistem de tip multiprocesor, multicore sau cu suport de hyperthreading. Pentru a transforma o interogare clasică într-una paralelizată, trebuie adăugată specificarea **AsParallel** la sursa de date peste care se execută interogarea. Mai exact, plecând de la interogarea LINQ:

```
var result =  
    from x in [sursa_de_date]  
    where [conditie]  
    select [ceva]
```

se obține interogarea PLINQ:

```
var result =  
    from x in [sursa_de_date].AsParallel()  
    where [conditie]  
    select [ceva]
```

Echivalent, se poate face transformarea folosind metode LINQ:

```
//varianta fara paralelism  
var result = [sursa_de_date].Where(x => [conditie]).Select(x => [ceva]);  
//varianta cu paralelism  
var result = [sursa_de_date].AsParallel().Where(x => [conditie]).  
    Select(x => [ceva]);
```

Prin adăugarea acestei metode `AsParallel()` se folosesc metode din clasa `ParallelEnumerable`, în timp ce în LINQ-ul clasic se folosesc metode din clasa `Enumerable`.

Prezentăm următorul exemplu preluat din [9]<sup>1</sup>:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace TestPLINQ
{
    class Program
    {
        static void Main()
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();
            DoIt();
            sw.Stop();
            Console.WriteLine("Elapsed = {0} milliseconds",
                sw.ElapsedMilliseconds.ToString());
        }

        private static bool isPrime(int p)
        {
            int upperBound = (int)Math.Sqrt(p);
            for (int i = 2; i <= upperBound; i++)
            {
                if (p % i == 0) return false;
            }
            return true;
        }

        static void DoIt()
        {
            IEnumerable<int> arr = Enumerable.Range(2, 3000000);
```

---

<sup>1</sup>Desigur, o variantă mai eficientă pentru determinarea numerelor prime de la 2 la  $n$  rămâne **Ciurul lui Eratostene**. Și acesta poate fi paralelizat.

```

        var q =
            from n in arr
            where isPrime(n)
            select n;
        IList<int> list = q.ToList();
        Console.WriteLine(list.Count.ToString());
    }
}

```

Pentru un sistem oarecare, timpul mediu de rulare este de aproximativ 23 secunde; rescriind interogarea pentru obținerea variabilei `q` astfel:

```

var q =
    from n in arr.AsParallel()
    where isPrime(n)
    select n.ToString();

```

timpul mediu obținut este de aproximativ 13 secunde.

Se pot configura aspecte precum gradul de paralelism, controlul ordinii, opțiuni de utilizare de buffere, dacă anumite porțiuni să se ruleze secvențial etc. prin folosirea metodelor de extensie precum `AsOrdered`, `AsUnordered` sau prin folosirea enumerării `ParallelQueryMergeOptions`.

Pentru controlul gradului de paralelism se poate folosi *WithDegreeOfParallelism* după *AsParallel()*:

```

var query = from n
              in arr.AsParallel().WithDegreeOfParallelism(2)
              where isPrime(n)
              select n;

```

O arie înrudită de explorare este Task Parallel Library.

## 13.2 Parametri cu nume și parametri opționali

Parametrii opționali permit precizarea unor valori implicite pentru parametrii unor metode; dacă aceștia nu sunt precizați la apel, atunci valorile trimise metodei sunt cele declarate implicit.

Exemplu:

```

class Program
{

```

```

static void Main(string[] args)
{
    MyMethod(3);
    MyMethod(3, 4);
}

private static void MyMethod(int p, int q = 100)
{
    Console.WriteLine("p= {0}, q={1}", p, q);
}
}

```

Deși avem o singură metodă, aceasta suportă cele două apeluri. Se poate chiar să avem mai mulți de un parametru cu valoare implicită:

```

private static void MyMethod(int p=1, int q = 100)
{
    Console.WriteLine("p= {0}, q={1}", p, q);
}

```

În cazul în care avem măcar un astfel de parametru cu valoarea implicită, acesta trebuie să fie prezent după parametrii cu valori obligatorii. Astfel, încercarea de a scrie:

```
private void MyMethod(int p=1, int q){...}
```

duce la apariția erorii de compilare:

Optional parameters must appear after all required parameters

Valorile furnizate pentru parametrii cu valori implicite trebuie să fie constante sau să aibă valori de forma `default(T)`, unde `T` este tipul parametrului.

În ceea ce privește folosirea parametrilor cu nume, să presupunem că avem următoarea metodă:

```
public void M(int x, int y = 5, int z = 7){...}
```

Dacă vrem ca la un apel să nu precizăm valoarea lui `y`, dar să o precizăm pe a lui `z`, am fi tentați să folosim:

```
M(1, , -1)
```

ceea ce în cazul în care ar fi permis, ar duce la un cod greu de citit: abilitatea de numărare a virgulelor ar fi crucială. În locul acestei variante, s-a introdus posibilitatea de a preciza parametrii prin nume:

```

M(1, z:3);
//sau
M(x:1, z:3);
//sau chiar:
M(z:3, x:1);

```

Ordinea de evaluare a expresiilor date pentru parametri este dată de ordinea de precizare a numelor parametrilor, deci în ultimul exemplu expresia “3” este considerată înainte de expresia “1”. Parametrii opționali și cu nume se pot folosi și pentru constructori sau indexatori.

## 13.3 Tipuri de date dinamice

Tipurile dinamice permit tratarea unui obiect fără a fi crispați de proveniența acestuia: obiect creat clasic, sau prin COM sau prin reflectare. Unui astfel de obiect *i* se pot transmite mesaje (=apeluri de metode), iar legitimitatea acestor apeluri este verificată la rulare. Altfel zis, pentru tipurile de date dinamice se renunță la tipizarea statică specifică platformelor .NET de versiune anterioară, dar cu riscul de a primi erori doar la rularea aplicației.

Acest tip de date dinamic se declară prin cuvântul cheie **dynamic**. Plecând de la un astfel de obiect, se pot apela diferite metode:

```

dynamic x = MyMethod();
x.AnotherMethod(3);

```

La rulare se verifică dacă tipul de date aferent variabilei **x** are metoda **AnotherMethod** care să permită apel cu un parametru de tip întreg.

Exemplu:

```

static void Main(string[] args)
{
    dynamic x = "abc";
    x = 3;
    Console.WriteLine(x.CompareTo(10)); //metoda CompareTo este din
    //tipul System.Int32
}

```

Remarcăm din exemplul anterior că tipul unei variabile dinamice nu este setat odată pentru totdeauna. Alt exemplu este:

```

class ExampleClass
{

```

```
public ExampleClass() { }
public ExampleClass(int v) { }

public void exampleMethod1(int i) { }

public void exampleMethod2(string str) { }
}

//....
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following line causes a compiler error
    // one parameter.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

Se pot declara ca fiind dinamici și parametrii unei metode:

```
public static void Log(dynamic x)
{
    Console.WriteLine(x.ToString());
}

static void Main(string[] args)
{
    var x = 3;
    string y = "abc";

    Log(x);
    Log(y);
}
```

```
}
```

## 13.4 ExpandoObject

Notă: prezentarea din această secțiune este preluată din MSDN, [ExpandoObject Class](#)

ExpandoObject este un tip de date ale cărui instanțe pot fi adăugate și șterse dinamic la rulare. De asemenea, valorile acestor membri pot fi setate la rulare. Scopul acestui tip este implementarea conceptului de obiect dinamic.

Pentru instanțiere se folosește:

```
dynamic sampleObject = new ExpandoObject();
```

Adăugarea unei noi proprietăți se face cu:

```
sampleObject.test = "Dynamic Property";  
Console.WriteLine(sampleObject.test);  
Console.WriteLine(sampleObject.test.GetType());  
// Dynamic Property  
// System.String
```

Pentru adăugarea unei metode care face incrementarea unui camp – de asemenea adăugat dinamic – scriem:

```
sampleObject.number = 10;  
sampleObject.Increment = (Action)(() => { sampleObject.number++; });  
  
// Before calling the Increment method.  
Console.WriteLine(sampleObject.number);  
  
sampleObject.Increment();  
  
// After calling the Increment method.  
Console.WriteLine(sampleObject.number);  
// This code example produces the following output:  
// 10  
// 11
```

Trimiterea ca parametru a unui astfel de obiect se face cu `dynamic`.

Având în vedere că tipul `ExpandoObject` implementează interfața `IDictionary<String, Object>`, putem enumera la rulare conținutul unui obiect:

```
dynamic employee = new ExpandoObject();
employee.Name = "John Smith";
employee.Age = 33;

foreach (var property in (IDictionary<String, Object>)employee)
{
    Console.WriteLine(property.Key + ": " + property.Value);
}
// This code example produces the following output:
// Name: John Smith
// Age: 33
```

## 13.5 COM Interop

COM Interop este o facilitate existentă în versiuni mai vechi ale lui .NET framework care permite codului managed să apeleze cod unmanaged Component Object Model, cum ar fi aplicațiile Word sau Excel. De exemplu, pentru utilizarea unei celule de Excel, varianta veche de cod ce trebuia scrisă era:

```
((Excel.Range)excel.Cells[1, 1]).Value2 = "Hello";
```

pe când în .NET 4 a se scrie mai inteligibil astfel:

```
excel.Cells[1, 1].Value = "Hello";
```

sau în loc de

```
Excel.Range range = (Excel.Range)excel.Cells[1, 1];
```

se scrie:

```
Excel.Range range = excel.Cells[1, 1];
```

Exemplu:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Excel = Microsoft.Office.Interop.Excel;
//in this project: add a reference to Microsoft.Office.Interop.Excel
```



```

namespace TestOffice
{
    class Program
    {
        static void Main(string[] args)
        {
            var excel = new Excel.Application();
            // Make the object visible.
            excel.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a particular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excel.Workbooks.Add();

            excel.Cells[1, 1].Value = "Hello";

            var processes = Process.GetProcesses()
                .OrderByDescending(p => p.WorkingSet64)
                .Take(10);
            int i = 2;
            foreach (var p in processes)
            {
                excel.Cells[i, 1].Value = p.ProcessName; // no casts
                excel.Cells[i, 2].Value = p.WorkingSet64; // no casts
                i++;
            }

            Excel.Range range = excel.Cells[1, 1]; // no casts
            dynamic chart = excel.ActiveWorkbook.Charts.
                Add(After: excel.ActiveSheet); // named and optional argument
            chart.ChartWizard(
                Source: range.CurrentRegion,
                Title: "Memory Usage in " + Environment.MachineName); //named+optional
            chart.ChartStyle = 45;
            chart.CopyPicture(Excel.XlPictureAppearance.xlScreen,
                Excel.XlCopyPictureFormat.xlBitmap,
                Excel.XlPictureAppearance.xlScreen);
        }
    }
}

```

```
    }
}
```

## 13.6 Covarianță și contravarianță

Să presupunem că avem secvența de cod:

```
var strList = new List<string>();
List<object> objList = strList;//referinta catre aceeași lista
```

Linia a doua, dacă ar fi permisă, ar predispute la erori, deoarece s-ar permite mai departe:

```
objList.Add(new MyClass());
```

deci s-ar încălca punctul de plecare pentru colecția **strList**: elementele ar trebui să fie toate de tip **String**. Ca atare, acest lucru nu ar avea sens să fie permis.

Începând cu .NET 4, se permite însă conversia către *interfețe generice sau delegați generici* pentru care tipul generic este mai general decât argumentul generic dinspre care se face conversie. De exemplu, se poate scrie:

```
IEnumerable<object> myCollection = strList;
```

Aceasta este covarianța și se obține prin declarațiile:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IEnumerator
{
    bool MoveNext();
    T Current { get; }
    void Reset();
}
```

unde cuvântul **out** are un alt sens decât la transmiterea de parametri: în C# 4.0, în acest context, semnifică faptul că tipul **T** poate să apară doar ca rezultat de return al unei metode din interfață. Interfața **IEnumerable** devine astfel covariantă în **T** și se poate face conversie de la **IEnumerable<B>** la **IEnumerable<A>** pentru tipul **B** derivat din **A**.

Acest lucru este util pentru o situație de genul:

```
var result = strings.Union(objects);
```

unde se face reuniune între o colecție de string-uri și una de obiecte.

Contravarianța permite conversii în sens invers ca la covarianță. De exemplu, prin declarația:

```
public interface IComparer<in T>
{
    public int Compare(T left, T right);
}
```

prin care se arată contravarianță a tipului `T`, deci se poate face ca un `IComparer<object>` să fie considerat drept un `IComparer<String>`. Are sens, deoarece dacă un `IComparer` poate să compare orice două obiecte, atunci cu siguranță poate să compare și două string-uri.

## 13.7 Clasele BigInteger și Complex

În .NET 4 s-a introdus spațiul de nume `System.Numerics` care conține structura `BigInteger`, specializată în lucrul cu numere întregi mari: operații aritmetice și pe biți.

```
BigInteger n = 1;
for (int i = 1; i <= 100; i++)
{
    n = n * i;
}
Console.WriteLine("100! are {0} cifre: {1}",
    n.ToString().Length.ToString(), n.ToString());
```

cu rezultatul:

100! are 158 cifre: 93326215443944152681699238856266700490715  
9682643816214685929638952175999932299156089414639761565182862  
53697920827223758251185210916864000000000000000000000

În același spațiu de nume există și structura `Complex`, specializată pe lucrul cu numere complexe:

```
Complex c = new Complex(3, 5);
Console.WriteLine(c.ToString());
var c2 = c * c;
Console.WriteLine(c2.ToString());
```

## 13.8 Clasa Tuple

În spațiul de nume `System` există clasa `Tuple` ce permite crearea de obiecte tuplu cu date structurate. Clasa pune la dispoziție metode generice `Create` care permit crearea de tupluri conținând până la 8 obiecte:

```
var tuple3 = Tuple.Create("New York", 32.68, 51.87);
Console.WriteLine("{0} {1} {2}", tuple3.Item1,
    tuple3.Item2, tuple3.Item3);

//This code is equivalent to the following call to the
//Tuple<T1, T2, T3>.Tuple<T1, T2, T3> class constructor.
var tuple3 = new Tuple<string, double, double>
    ("New York", 32.68, 51.87);

//tuple3.Item1 = "Boston";
//compile error: Property or indexer
//'System.Tuple<string,double,double>.Item1' cannot
//be assigned to -- it is read only
```

## Curs 14

# Fluxuri

Pentru aplicațiile reale, datele care se prelucrează nu se mai preiau din tastatură, ci se accesează din fișiere de date. C#, precum alte limbaje anterioare, pune la dispoziție o abstractizare numită “flux”<sup>1</sup> care permite manipularea datelor, indiferent de sursa acestora. Într-un astfel de flux, pachete de date urmează unele după altele.

În C# fișierele și directoarele se manipulează prin intermediul unor clase predefinite. Acestea permit crearea, redenumirea, manipularea, ștergerea fișierelor și a directoarelor. Manipularea *conținutului* fișierelor se face prin intermediul stream-urilor cu sau fără buffer; de asemenea există un suport puternic pentru stream-uri asincrone (prelucrarea unui fișier se face de către un fir de execuție creat automat). Datorită abstractizării, nu există diferențe mari între lucrul cu fișiere aflate pe discul local și datele aflate pe rețea; ca atare se va vorbi despre fluxuri bazate pe protocoale TCP/IP sau web. În sfârșit, *serializarea* este legată de fluxuri, întrucât permit “îngețarea” unui obiect (care poate fi urmată de transportul lui pe rețea).

### 14.1 Sistemul de fișiere

Clasele care se folosesc pentru manipularea fișierelor și a directoarelor se află în spațiul de nume *System.IO*. Funcționalitatea lor este aceeași cu a comenzilor disponibile într-un sistem de operare: creare, ștergere, redenumire, mutare de fișiere sau directoare, listarea conținutului unui director, listarea atributelor sau a diferiților timpi pentru fișiere sau directoare etc.

Clasele pe care vom folosi sunt: *Directory*, *DirectoryInfo*, *File*, *FileInfo*.

---

<sup>1</sup>Engl: Stream

### 14.1.1 Lucrul cu directoarele: clasele *Directory* și *DirectoryInfo*

Clasa *Directory* conține metode statice pentru crearea, mutarea, explorarea directoarelor. Clasa *DirectoryInfo* conține doar membri nestatici și permite aflarea diferitelor informații pentru un director anume.

Tabelul 14.1 conține principalele metode ale clasei *Directory*, iar tabelul 14.2 conține metodele notabile din clasa *DirectoryInfo*.

Tabelul 14.1: Metode ale clasei *Directory*.

Metoda	Explicație
CreateDirectory()	Creează directoarele și subdirectoarele specificate prin parametru
Delete()	Șterge un director și conținutul său
Exists()	Returnează <i>true</i> dacă stringul specificat reprezintă numele unui director existent
GetCreationTime() SetCreationTime()	Returnează / setează data și timpul creării unui director
GetCurrentDirectory() SetCurrentDirectory()	returnează / setează directorul curent
GetDirectories()	Returnează un șir de nume de subdirectoare
GetDirectoryRoot()	Returnează numele rădăcinii unui director specificat
GetFiles()	Returnează un șir de string-uri care conține numele fișierelor din directorul specificat
GetLastAccessTime() SetLastAccessTime()	returnează / setează timpul ultimului acces pentru un director
GetLastWriteTime() SetLastWriteTime()	Returnează / setează timpul când directorul specificat a fost ultima oară modificat
GetLogicalDrives()	Returnează numele tuturor unităților logice sub forma drive:\
GetParent()	Returnează directorul curent pentru calea specificată
Move()	Mută un director (cu conținut) într-o cale specificată

Tabelul 14.2: Metode și proprietăți ale clasei *DirectoryInfo*.

Metoda sau proprietate	Explicație
Attributes	Returnează sau setează attributele fișierului curent

Tabelul 14.2(continuare)

Metoda sau proprietate	Explicație
CreationTime	Returnează sau setează timpul de creare al fișierului curent
Exists	<i>true</i> dacă directorul există
Extension	Extensia directorului
FullName	Returnează calea absolută a fișierului sau a directorului
LastAccessTime	Returnează sau setează timpul ultimului acces
LastWriteTime	Returnează sau setează timpul ultimei scrieri
Parent	Directorul părinte al directorului specificat
Root	Rădăcina căii corespunzătoare
Create()	Creează un director
CreateSubdirectory()	Creează un subdirector în calea specificată a
Delete()	Șterge un <i>DirectoryInfo</i> și conținutul său
GetDirectories()	Returnează un vector de tip <i>DirectoryInfo</i> cu subdirectoare
GetFiles()	Returnează lista fișierelor din director
MoveTo()	Mută un <i>DirectoryInfo</i> și conținutul său într-un nou loc

Exemplul următor folosește clasa *DirectoryInfo* pentru a realiza explorarea recursivă a unui director cu enumerarea tuturor subdirectoarelor conținute. Se creează un obiect de tipul pomenit pe baza numelui de director de la care se va începe explorarea. O metodă va afișa numele directorului la care s-a ajuns, după care se apelează recursiv metoda pentru fiecare subdirector (obținut via *GetDirectories()*).

```
using System;
using System.IO;
class Tester
{
    public static void Main( )
    {
        Tester t = new Tester( );
        //choose the initial subdirectory
        string theDirectory = @"c:\WinNT";
        // call the method to explore the directory,
        // displaying its access date and all
        // subdirectories
```

```

        DirectoryInfo dir = new DirectoryInfo(theDirectory);
        t.ExploreDirectory(dir);
        // completed. print the statistics
        Console.WriteLine(@"\n\n{0} directories found.\n", dirCounter);
    }
    // Set it running with a DirectoryInfo object
    // for each directory it finds, it will call
    // itself recursively
    private void ExploreDirectory(DirectoryInfo dir)
    {
        indentLevel++; // push a directory level
        // create indentation for subdirectories
        for (int i = 0; i < indentLevel; i++)
            Console.Write(" "); // two spaces per level
        // print the directory and the time last accessed
        Console.WriteLine("[{0}] {1} [{2}]\n",
            indentLevel, dir.Name, dir.LastAccessTime);
        // get all the directories in the current directory
        // and call this method recursively on each
        DirectoryInfo[] directories = dir.GetDirectories();
        foreach (DirectoryInfo newDir in directories)
        {
            dirCounter++; // increment the counter
            ExploreDirectory(newDir);
        }
        indentLevel--; // pop a directory level
    }
    // static member variables to keep track of totals
    // and indentation level
    static int dirCounter = 1;
    static int indentLevel = -1; // so first push = 0
}

```

### 14.1.2 Lucrul cu fişierele: clasele *FileInfo* şi *File*

Un obiect *DirectoryInfo* poate de asemenea să returneze o colecţie a tuturor fişierelor conţinute, sub forma unor obiecte de tip *FileInfo*. Înrudită cu clasa *FileInfo* (care are membri nestatici) este clasa *File* (care are doar membri statici). Tabelele 14.3 şi 14.4 conţin metodele pentru fiecare clasă:



Tabelul 14.3: Metode ale clasei *File*.

Metoda	Explicație
AppendText()	Creează un obiect <i>StreamWriter</i> care adaugă text la fișierul specificat
Copy()	Copiază un fișier existent într-un alt fișier
Create()	Creează un fișier în calea specificată
CreateText()	Creează un <i>StreamWriter</i> care scrie un nou fișier text
Delete()	Șterge fișierul specificat
Exists()	Returnează true dacă fișierul corespunzător există
GetAttributes() SetAttributes()	Returnează / setează <i>FileAttributes</i> pentru fișierul specificat
GetCreationTime() SetCreationTime()	Returnează / setează data și timpul creării pentru fișierul specificat
GetLastAccessTime() SetLastAccessFile()	Returnează sau setează timpul ultimului acces la fișier
GetLastWriteTime() SetLastAccessTime()	Returnează / setează timpul ultimei modificări a fișierului
Move()	Mută un fișier la o nouă locație; poate fi folosit pentru redenumire
OpenRead()	Metodă returnând un <i>FileStream</i> pe un fișier
OpenWrite()	Crează un <i>Stream</i> de citire / scriere

Tabelul 14.4: Metode și proprietăți ale clasei *FileInfo*.

Metoda sau proprietatea	Explicație
Attributes	Returnează sau setează atributele fișierului curent
CreationTime	Returnează sau setează timpul de creare al fișierului curent
Directory	Returnează o instanță a directorului curent
Exists	true dacă fișierul există
Extension	Returnează extensia fișierului
FullName	Calea absolută până la fișierul curent
LastAccessTime	Returnează sau setează timpul ultimului acces
LastWriteTime	Returnează sau setează timpul când s-a modificat ultima oară fișierul curent
Length	Returnează dimensiunea fișierului
Name	Returnează numele instanței curente
AppendText	Crează un <i>StreamWriter</i> care va permite adăugarea

Tabelul 14.4 (continuare)

Metoda sau proprietatea	Explicație
	la fișier
CopyTo()	Copiează fișierul curent într-un alt fișier
Create()	Crează un nou fișier
Delete()	Șterge un fișier
MoveTo()	Mută un fișier la o locație specificată; poate fi folosită pentru redenumire
OpenRead() OpenText() OpenWrite()	Crează un fișier read-only respectiv StreamReader(text) sau <i>FileStream</i> (read-write)

Exemplul anterior este modificat pentru a afișa informații despre fișiere: numele, dimensiunea, data ultimei modificări:

```
using System;
using System.IO;
class Tester
{
    public static void Main( )
    {
        Tester t = new Tester( );
        // choose the initial subdirectory
        string theDirectory = @"c:\WinNT";
        // call the method to explore the directory,
        // displaying its access date and all
        // subdirectories
        DirectoryInfo dir = new DirectoryInfo(theDirectory);
        t.ExploreDirectory(dir);
        // completed. print the statistics
        Console.WriteLine(@"\n\n{0} files in {1} directories found.\n",
            fileCounter,dirCounter);
    }
    // Set it running with a DirectoryInfo object
    // for each directory it finds, it will call
    // itself recursively
    private void ExploreDirectory(DirectoryInfo dir)
    {
        indentLevel++; // push a directory level
        // create indentation for subdirectories
        for (int i = 0; i < indentLevel; i++)
```

```

    Console.Write("  "); // two spaces per level
    // print the directory and the time last accessed
    Console.WriteLine("[{0}] {1} [{2}]\n", indentLevel, dir.Name,
        dir.LastAccessTime);
    // get all the files in the directory and
    // print their name, last access time, and size
    FileInfo[] filesInDir = dir.GetFiles( );
    foreach (FileInfo file in filesInDir)
    {
        // indent once extra to put files
        // under their directory
        for (int i = 0; i < indentLevel+1; i++)
            Console.Write("  "); // two spaces per level
        Console.WriteLine("{0} [{1}] Size: {2} bytes", file.Name, file.LastWriteTime,
            file.Length);
        fileCounter++;
    }
    // get all the directories in the current directory
    // and call this method recursively on each
    DirectoryInfo[] directories = dir.GetDirectories( );
    foreach (DirectoryInfo newDir in directories)
    {
        dirCounter++; // increment the counter
        ExploreDirectory(newDir);
    }
    indentLevel--; // pop a directory level
}
// static member variables to keep track of totals
// and indentation level
static int dirCounter = 1;
static int indentLevel = -1; // so first push = 0
static int fileCounter = 0;
}

```

Exemplul următor nu introduce clase noi, ci doar exemplifică crearea unui director, copierea de fișiere în el, ștergerea unora dintre ele și în final ștergerea directorului:

```

using System;
using System.IO;
class Tester
{

```

```

public static void Main( )
{
    // make an instance and run it
    Tester t = new Tester( );
    string theDirectory = @"c:\test\media";
    DirectoryInfo dir = new DirectoryInfo(theDirectory);
    t.ExploreDirectory(dir);
}
// Set it running with a directory name
private void ExploreDirectory(DirectoryInfo dir)
{
    // make a new subdirectory
    string newDirectory = "newTest";
    DirectoryInfo newSubDir =
    dir.CreateSubdirectory(newDirectory);
    / get all the files in the directory and
    // copy them to the new directory
    FileInfo[] filesInDir = dir.GetFiles( );
    foreach (FileInfo file in filesInDir)
    {
        string fullName = newSubDir.FullName + "\\\" + file.Name;
        file.CopyTo(fullName);
        Console.WriteLine("{0} copied to newTest", file.FullName);
    }
    // get a collection of the files copied in
    filesInDir = newSubDir.GetFiles( );
    // delete some and rename others
    int counter = 0;
    foreach (FileInfo file in filesInDir)
    {
        string fullName = file.FullName;
        if (counter++ %2 == 0)
        {
            file.MoveTo(fullName + ".bak");
            Console.WriteLine("{0} renamed to {1}",
            fullName,file.FullName);
        }
        else
        {
            file.Delete( );
            Console.WriteLine("{0} deleted.",

```

```

fullName);
    }
}
newSubDir.Delete(true); // delete the subdirectory
}
}

```

## 14.2 Citirea și scrierea datelor

Clasele disponibile pentru lucrul cu stream-uri sunt:

Tabelul 14.5: Clase pentru lucrul cu stream-uri

Clasa	Descriere
Stream	Manipulare generică a unei secvențe de octeți; clasă abstractă
BinaryReader	Citește tipuri de date primitive ca valori binare într-o codificare specifică
BinaryWriter	Scrie tipuri de date primitive într-un flux binar; de asemenea scrie string-uri folosind o anumită codificare
BufferedStream	Atașează un buffer unui stream de intrare / ieșire. Clasă <i>sealed</i>
FileStream	Atașează un stream unui fișier, permițând operații sincrone sau asincrone de citire și scriere.
MemoryStream	Crează un stream pentru care citirea / stocarea de date se face în memorie
NetworkStream	Creează un stream folosind TCP/IP
TextReader	Permite citire de caractere, în mod secvențial. Clasă abstractă.
TextWriter	Permite scriere secvențială într-un fișier. Clasă abstractă.
StreamReader	Implementează o clasă <i>TextReader</i> care citește caractere dintr-un stream folosind o codificare particulară
StreamWriter	Implementează o clasă <i>TextWriter</i> care scrie caractere într-un stream folosind o codificare particulară
StringReader	Implementează un <i>TextReader</i> care citește dintr-un string
StringWriter	Scrie informație într-un string. Informația este stocată într-un <i>StringBuilder</i>

### 14.2.1 Clasa *Stream*

Clasa *Stream* este o clasă abstractă din care se derivează toate celelalte clase de lucru cu stream-uri. Metodele conținute permit citire de octeți,

închidere, golire de buffer etc. Pe lângă acestea, clasa *Stream* permite atât operații sincrone, cât și asincrone. Într-o operație de intrare / ieșire sincronă, dacă se începe o operație de citire sau scriere dintr-un / într-un flux, atunci programul va trebui să aștepte până când această operație se termină. Sub platforma .NET se poate face însă operație de intrare / ieșire în mod asincron, astfel permițând altor fire de execuție să se execute.

Metodele folosite pentru a începe o astfel de intrare asincronă sunt: *BeginRead()*, *BeginWrite()*, *EndRead()*, *EndWrite()*. O dată ce o astfel de operație se termină, o metodă callback se va executa automat.

O listă a metodelor și proprietăților care sunt definite în clasa *Stream* este dată în tabelul 14.6.

Tabelul 14.6: Metode și proprietăți ale clasei *Stream*

Clasa	Descriere
<i>BeginRead()</i>	Începe o citire asincronă
<i>BeginWrite()</i>	Începe o scriere asincronă
<i>Close()</i>	Închide stream-ul curent și eliberează resursele asociate cu el (socket-uri, file handles etc)
<i>EndRead()</i>	Așteaptă pentru o terminare de citire asincronă.
<i>EndWrite()</i>	Așteaptă pentru o terminare de scriere asincronă.
<i>Flush()</i>	Când este suprascrisă în tr-o clasă derivată, golește bufferele asociate stream-ului curent și determină scrierea lor.
<i>Read()</i>	Când este suprascrisă într-o clasă derivată, citește o secvență de octeți și incrementează indicatorul de poziție curentă în stream
<i>ReadByte()</i>	Citește un byte din stream și incrementează indicatorul de poziție curent; dacă este la sfârșit de fișier, returnează -1
<i>Seek()</i>	Când este suprascrisă într-o clasă derivată, setează poziția curentă în interiorul stream-ului
<i>SetLength()</i>	Când este suprascrisă într-o clasă derivată, setează lungimea stream-ului curent
<i>Write()</i>	Când este suprascrisă într-o clasă derivată, scrie o secvență de octeți în stream-ul curent și incrementează corespunzător indicatorul poziției curente în stream
<i>WriteByte()</i>	Scrie un byte la poziția curentă din stream și incrementează indicatorul de poziție curentă
<i>CanRead()</i>	Când este suprascrisă într-o clasă derivată, returnează o valoare care indică dacă stream-ul curent poate fi citit
<i>CanWrite()</i>	Când este suprascrisă într-o clasă derivată, returnează

Tabelul 14.6 (continuare)

Metoda	Descriere
	o valoare care indică dacă stream-ul curent suportă scriere
CanSeek	Când este suprascrisă într-o clasă derivată, returnează o valoare care indică dacă se poate face poziționare aleatoare în stream-ul curent
Length	Când este suprascrisă într-o clasă derivată, returnează dimensiunea în octeți a fișierului
Position	Când este suprascrisă într-o clasă derivată, returnează sau setează poziția curentă în interiorul stream-ului

### 14.2.2 Clasa *FileStream*

Există mai multe metode de obținere a unui obiect de tip *FileStream*. Clasa prezintă nouă constructori supraîncărcați. Enumerarea *FileMode* este folosită pentru a se specifica modul de deschidere a unui stream: (*Append*, *Create*, *CreateNew*, *Open*, *OpenOrCreate*, *Truncate*).

Exemplu: mai jos se creează un fișier nou (dacă nu există) sau se suprascrie unul existent:

```
FileStream f = new FileStream( @"C:\temp\a.dat", FileMode.Create );
```

De asemenea se mai poate obține o instanță a clasei *FileStream* din clasa *File*:

```
FileStream g = File.OpenWrite(@"c:\temp\test.dat");  
//deschidere doar pentru citire
```

Asemănător, se poate folosi o instanță a clasei *FileInfo*:

```
FileInfo fi = new FileInfo(@"c:\temp\test.dat");  
FileStream fs = fi.OpenRead();  
//deschidere doar pentru citire
```

### 14.2.3 Clasa *MemoryStream*

Un *MemoryStream* își ia datele din memorie, văzută ca un vector de octeți. Există șapte constructori pentru această clasă, dar care pot fi grupați în două categorii.

Primul tip de constructor preia un array de octeți pentru care poate face citire și opțional scriere. Caracteristic este că tabloul nu va putea fi redimensionat:

```
byte[] b = {1, 2, 3, 4};
MemoryStream mem = new MemoryStream(b);
```

O altă variantă de constructor nu primește un vector de octeți, dar va putea scrie într-un tablou redimensionabil. Opțional, se specifică un *int* ca parametru al constructorului care determină dimensiunea inițială a tabloului de octeți. Datele sunt adăugate folosind metoda *Write()*:

```
using System;
using System.IO;

public class MemTest
{
    public static void Main()
    {
        MemoryStream mem = new MemoryStream();
        byte[] bs = {1, 2, 3, 4, 5, 6};
        mem.Write(bs, 0, bs.Length);
        mem.Seek(3, SeekOrigin.Begin);
        byte b = (byte)mem.ReadByte();
        Console.WriteLine("Value: {0}", b.ToString());
        //se va afisa 4
    }
}
```

E de preferat să se utilizeze obiecte de tip *MemoryStream* în scopul de a accesa informația din memoria RAM, mai degrabă decât de pe disc sau din rețea. De exemplu se poate încărca un fișier de pe disc în memorie, astfel încât analiza lui se poate face mai repede.

#### 14.2.4 Clasa *BufferedStream*

C# pune la dispoziție o clasă *BufferedStream* pentru a asigura o zonă tampon în cazul operațiilor de intrare-ieșire. Constructorul acestei clase primește o instanță a clasei *Stream*.

Exemplu:

```
FileStream fs = new FileStream(@"c:\temp\a.dat", FileMode.Open);
BufferedStream bs = new BufferedStream(fs);
```

De menționat aici metoda *Flush()*, care forțează golirea bufferului asociat stream-ului.



### 14.2.5 Clasele *BinaryReader* și *BinaryWriter*

Cele două clase sunt folosite pentru a accesa date mai complexe decât un byte: de exemplu, pentru manipularea datelor de tip boolean, sau Decimal, sau int cu semn pe 64 de biți.

Tabelul 14.7 conține metodele puse la dispoziție de către clasa *BinaryWriter*:

Tabelul 14.7: Metodele clasei *BinaryReader*

Metoda	Descriere
PeekChar()	Returnează următorul caracter disponibil, fără a avansa indicatorul de poziție curentă
Read()	Citește caractere din flux și avansează poziția curentă din acel stream
ReadBoolean()	Citește un Boolean din stream și avansează poziția curentă cu un byte
ReadBytes()	Citește un număr precizat de octeți într-un vector și avansează poziția curentă
ReadChar()	Citește următorul caracter și avansează poziția curentă corespunzător cu codificarea folosită pentru caracter
ReadChars()	Citește mai multe caractere într-un vector și avansează poziția curentă cu numărul de caractere dimensiunea de reprezentare pentru caracter
readDecimal()	Citește un decimal și avansează poziția curentă cu 16 octeți
ReadDouble()	Citește o variabilă în virgulă mobilă și avansează cu 8 octeți
ReadInt16()	Citește un întreg cu semn pe 16 biți și avansează cu 2 octeți
ReadInt32()	Citește un întreg cu semn pe 32 de biți și avansează cu 4 octeți
ReadInt64()	Citește un întreg cu semn pe 64 de biți și avansează cu 8 octeți
ReadSByte()	Citește un byte cu semn și avansează cu un byte
ReadSingle()	Citește o valoare în virgulă mobilă pe 4 octeți și avansează poziția curentă cu 4 octeți
ReadString()	Citește un string, prefixat cu lungimea sa, codificată ca un întreg reprezentat pe grupe de câte 7 biți (MSDN)
ReadUInt16	Citește un întreg fără semn reprezentat pe 16 biți și avansează cu 2 octeți
ReadUInt32	Citește un întreg fără semn reprezentat pe 32 de biți și avansează cu 4 octeți
ReadUInt64	Citește un întreg fără semn reprezentat pe 64 de biți și avansează cu 8 octeți

Clasa *BinaryWriter* are o metodă *Write()* supraîncărcată, care poate fi apelată pentru scrierea diferitelor tipuri de date. O mențione la scrierea de string-uri și de caractere / vectori de caractere: caracterele pot fi codificate în mai multe moduri (ex: ASCII, UNICODE, UTF7, UTF8), codificare care se poate transmite ca argument pentru constructor. A se vedea MSDN și RFC-urile de pe Internet.

### 14.2.6 Clasele *TextReader*, *TextWriter* și descendentele lor

Pentru manipularea șirurilor de caractere aflate în fișiere, dar nu numai, C# pune la dispoziție clasele abstracte *TextReader*, *TextWriter*. Clasa *TextReader* are subclasele neabstracte *StreamReader* și *StringReader*. Clasa *TextWriter* are subclasele neabstracte *StreamWriter*, *StringWriter*, *System.Web.HttpWriter*, *System.Web.UI.HtmlTextWriter*, *System.CodeDom.Compiler.IndentedTextWriter*.

Descrieri și exemple sunt date mai jos:

1. Clasele *StreamReader* și *StreamWriter* - sunt folosite pentru a citi sau scrie șiruri de caractere. Un obiect de tip *StreamReader* se poate obține via un constructor:

```
StreamReader sr = new StreamReader(@"C:\temp\siruri.txt");
```

sau pe baza unui obiect de tip *FileInfo*:

```
FileInfo fi = new FileInfo(@"c:\temp\siruri.txt");
StreamReader sr = fi.OpenText();
```

Obiectele de tip *StreamReader* pot citi câte o linie la un moment dat folosind metoda *ReadLine()*.

Exemplu:

```
using System;
using System.IO;
class Test
{
    static void Main()
    {
        StreamReader sr = new StreamReader(@"c:\temp\siruri.txt");
        String line;
        do
```

```

    {
        line = sr.ReadLine();
        Console.WriteLine(line);
        //daca line==null, atunci se va afisa o linie vida
    }while( line!=null);
}
}

```

2. Clasele *StringReader* și *StringWriter* - permit atașarea unor fluxuri la șiruri de caractere, folosite pe post de surse de date.

Exemplu:

```

string myString = "1234567890";
StringReader sr = new StringReader(myString);

using System;
using System.IO;
using System.Xml;
class Test
{
    static void Main()
    {
        XmlDocument doc = new XmlDocument();
        String entry = "<book genre='biography'>> " +
            "ISBN='12345678'><title>Yeager</title>> " +
            "</book>";
        doc.LoadXml(entry); //salvare in doc. xml

        StringWriter writer = new StringWriter();
        doc.Save(writer);
        string strXml = writer.ToString();
        Console.WriteLine(strXml);
    }
}

```

va afișa:

```

<?xml version="1.0" encoding="utf-16">
<book genre="biography" ISBN="12345678">
    <title>Yeager</title>
</book>

```

În loc ca salvarea din documentul xml să se facă într-un fișier text, se face într-un obiect de tip `StringWriter()`, al cărui conținut se va afișa.

3. *IndentedTextWriter* definește metode care inserează tab-uri și păstrează evidența nivelului de indentare. Este folosit de deriuvări ale clasei *CodeDom*, folosită pentru generare de cod.
4. *HtmlTextWriter* scrie o secvență HTML pe o pagină Web. Este folosit de exemplu în script-uri C#, în cazul aplicațiilor ASP.NET
5. *HttpWriter* - prea puțină informație!!!

### 14.3 Operare sincronă și asincronă

Exemplele folosite până acum au folosit un mod de operare sincron, adică atunci când se face o operație de intrare/ieșire, întregul program este blocat până când se tranzitează toate datele specificate. Stream-urile C# permit și acces asincron, permițând altor fire de execuție să fie rulate. Pentru semnalarea începutului unei operații de citire sau scriere asincrone, se folosesc *BeginRead()* și *BeginWrite()*.

Metoda *BeginRead* are prototipul:

```
public override IAsyncResult BeginRead(
    byte[] array, int offset, int numBytes,
    AsyncCallback userCallback, object stateObject
);
```

Vectorul de bytes reprezintă bufferul în care se vor citi datele; al doilea și al treilea parametru determină byte-ul la care să se va scrie, respectiv numărul maxim de octeți care se vor citi. Al patrulea parametru este un delegat, folosit pentru a semnala (mai exact, a face niște prelucrări) sfârșitul citirii. Se poate transmite *null* pentru acest parametru, dar programul nu va fi notificat de terminarea citirii. Ultimul parametru este un obiect care va fi folosit pentru a distinge între această cerere de citire asincronă și altele.

```
using System;
using System.IO;
using System.Threading;
using System.Text;
public class AsyncIOTester
{
    private Stream inputStream;
```

```
// delegated method
private AsyncCallback myCallBack;
// buffer to hold the read data
private byte[] buffer;
// the size of the buffer
const int BufferSize = 256;
// constructor
AsynchIOTester( )
{
    // open the input stream
    inputStream =
    File.OpenRead(
    @"C:\test\source\AskTim.txt");
    // allocate a buffer
    buffer = new byte[BufferSize];
    // assign the call back
    myCallBack =
    new AsyncCallback(this.OnCompletedRead);
}
public static void Main( )
{
    // create an instance of AsynchIOTester
    // which invokes the constructor
    AsynchIOTester theApp =
    new AsynchIOTester( );
    // call the instance method
    theApp.Run( );
}
void Run( )
{
    inputStream.BeginRead(
    buffer, // holds the results
    0, // offset
    buffer.Length, // (BufferSize)
    myCallBack, // call back delegate
    null); // local state object
    // do some work while data is read
    for (long i = 0; i < 500000; i++)
    {
        if (i%1000 == 0)
        {
```

```

Console.WriteLine("i: {0}", i);
    }
}
// call back method
void OnCompletedRead(IAsyncResult asyncResult)
{
    int bytesRead =
        inputStream.EndRead(asyncResult);
    // if we got bytes, make them a string
    // and display them, then start up again.
    // Otherwise, we're done.
    if (bytesRead > 0)
    {
        String s =
            Encoding.ASCII.GetString(buffer, 0, bytesRead);
        Console.WriteLine(s);
        inputStream.BeginRead(
            buffer, 0, buffer.Length, myCallBack, null);
    }
}
}

```

Ieșirea ar fi:

```

i: 47000
i: 48000
i: 49000
Date: January 2001
From: Dave Heisler
To: Ask Tim
Subject: Questions About O'Reilly
Dear Tim,
I've been a programmer for about ten years. I had heard of
O'Reilly books, then...
Dave,
You might be amazed at how many requests for help with
school projects I get;
i: 50000
i: 51000
i: 52000

```

Cele două fire de execuție lucrează deci concurent.

## 14.4 Stream-uri Web

C# conține clase gândite pentru a ușura interoperarea cu web-ul. Aducerea informației de pe web se face în doi pași: primul pas constă în a face o cerere de conectare la un server; dacă cererea se poate face, atunci în pasul al doilea constă în obținerea informației propriu-zise de la server. Cei doi pași se fac respectiv cu clasele *HttpRequest*, respectiv *HttpResponse*

Un obiect *HttpRequest* poate fi obținut prin metoda statică *Create()* din clasa *WebRequest*:

```
string page = "http://www.cetus-links.org/index.html";
HttpRequest webRequest = (HttpRequest)WebRequest.Create(page);
```

Pe baza obiectului *HttpRequest* obținut se va crea un obiect *HttpResponse*:

```
HttpResponse webResponse = (HttpResponse)webRequest.GetResponse();
```

În final, se obține un stream prin metoda *GetResponseStream()*:

```
StreamReader streamReader =
    new StreamReader(webResponse.GetResponseStream(), Encoding.ASCII);
```

## 14.5 Serializarea

Serializarea reprezintă posibilitatea de a trimite un obiect printr-un stream. Pentru aceasta, C# folosește metodele *Serialize()* și *Deserialize()* ale clasei *BinaryFormatter*.

Metoda *Serialize()* are nevoie de 2 parametri: un stream în care să scrie și obiectul pe care să îl serializeze. Metoda de deserializare cere doar un stream din care să citească, și din care să refacă un obiect (care poate fi convertit la tipul corespunzător).

### 14.5.1 Crearea unui obiect serializabil

Pentru ca o clasă definită de utilizator să suport serializarea, este nevoie de a preciza atributul *[Serializable]* în fața declarației clasei respective, atribut definit în clasa *System.SerializableAttribute*. Tipurile primitive sunt automat serializabile, iar dacă tipul definit de utilizator conține alte tipuri, atunci acestea la rândul lor trebuie să poată fi serializate.

Exemplu:

```
using System;
[Serializable]
public class BookRecord
{
    public String title;
    public int asin;
    public BookRecord(String title, int asin)
    {
        this.title = title;
        this.asin = asin;
    }
}
```

### 14.5.2 Serializarea

Codul pentru serializarea unui obiect de tipul declarat mai sus este:

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
public class SerializeObject
{
    public static void Main()
    {
        BookRecord book = new BookRecord(
            "Building Robots with Lego Mindstorms",
            1928994679);
        FileStream stream = new FileStream(@"book.obj",
            FileMode.Create);
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(stream, book);
        stream.Close();
    }
}
```

### 14.5.3 Deserializarea unui obiect

Deserializarea se face astfel:

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```



```
public class DeserializeObject
{
    public static void Main()
    {
        FileStream streamIn = new FileStream(
            @"book.obj", FileMode.Open);
        BinaryFormatter bf = new BinaryFormatter();
        BookRecord book =
            (BookRecord)bf.Deserialize(streamIn);
        streamIn.Close();
        Console.WriteLine(book.title + " " + book.asin);
    }
}
```

#### 14.5.4 Date tranziente

Uneori este nevoie ca anumite câmpuri ale unui obiect să nu fie salvate: parola unui utilizator, numărul de cont al unui client etc. Acest lucru se face specificând atributul *[NonSerialized]* pentru câmpul respectiv:

```
[NonSerialized] int secretKey;
```

#### 14.5.5 Operații la deserializare

Uneori este nevoie ca o deserializare să fie automat urmată de o anumită operație. De exemplu, un câmp care nu a fost salvat (tranzient) va trebui să fie refăcut în mod calculat. Pentru acest lucru, C# pune la dispoziție interfața *IDeserializationCallback*, pentru care trebuie să se implementeze metoda *OnDeserialization*. Această metodă va fi apelată automat de către CLR atunci când se va face deserializarea obiectului.

Exemplul de mai jos ilustrează acest mecanism:

```
using System;
using System.Runtime.Serialization;
[Serializable]
public class BookRecord: IDeserializationCallback
{
    public String title;
    public int asin;
    [NonSerialized] public int sales_rank;
    public BookRecord(String title, int asin)
    {
```

```
        this.title = title;
        this.asin = asin;
        sales_rank = GetSalesRank();
    }
    public int GetSalesRank()
    {
        Random r = new Random();
        return r.Next(5000);
    }
    public void OnDeserialization(Object o)
    {
        sales_rank = GetSalesRank();
    }
}
```

Mecanismul poate fi folosit în special atunci când serializarea unui anumit câmp, care ar duce la mult spațiu de stocare consumat, ar putea fi calculat în mod automat la deserializare (spațiu vs. procesor).

# Bibliografie

- [1] *C# in depth*, Jon Skeet, 2011, 2nd edition, Manning.
- [2] *Programming C#: Building Windows, Web, and RIA Applications for the .NET 4.0 Framework*, Ian Griffiths, Matthew Adams, Jesse Liberty, 2010, 4th edition, O'Reilly.
- [3] *Pro C# 2010 and the .NET 4 Platform*, Andrew Troelsen, 2010, 5th edition, APress.
- [4] *Introducere în limbajul C#*, Lucian Sasu, 2009, Editura Universității Transilvania.
- [5] *LINQ for Visual C# 2008*, Fabio Claudio Ferracchiati, Apress, 2008
- [6] *Core C# and .NET*, Stephen C. Perry, Prentice Hall, 2005
- [7] *C# Language Specification*, ECMA TC39/TG2, Octombrie 2002
- [8] *Professional ADO.NET Programming*, Wrox, 2001
- [9] *PLINQ*, Daniel Moth, 2009.