

1. Suportul pentru directoare și fișiere

La fel ca și limbajul C (revedi cursul de C), în C# salvarea și restaurarea datelor din fișiere implică noțiunea de *stream*. Un stream este o reprezentare abstractă a unui dispozitiv serial. Prin utilizarea acestui mecanism, programatorul nu este nevoit să cunoască structura hardware a echipamentului care stochează datele, accesul fiind făcut la nivel logic, prin intermediul fișierelor. Scrierea și citirea datelor din fișiere se face în mod secvențial, putând fi citiți și scriși fie octeți, fie caractere.

Citirea datelor se face prin intermediul unui stream de intrare. Cel mai des utilizat stream de intrare este tastatura. Dar mecanismul poate fi aplicat și altor echipamente periferice, sau fișierelor obișnuite.

Scrierea datelor se face prin intermediul unui stream de ieșire. Să ne reamintim ca streamul standard de ieșire este videoterminalul, dar pot fi definite și alte echipamente ca streamuri de ieșire, sau pur și simplu fișiere.

1.1 Clase pentru operații intrare-ieșire

Toate clasele implicate în transferul și stocarea datelor în fișiere sunt definite în spațiul de nume **System.IO**. Pentru a putea utiliza aceste clase, va trebui să specificăm la începutul fișierului sursă în care se declară clasele de lucru utilizarea acestui spațiu (dacă nu există) prin directiva

using System.IO;

Există mai multe clase conținute în acest spațiu de nume, clase care ne permit să manipulăm atât fișiere cât și directoare. Câteva din aceste clase sunt:

- **File** – este o clasă care expune un set de metode statice (să ne reamintim, *o metodă statică este apelată prin intermediul clasei și nu a unui obiect din clasa respectivă, fiind un atribut al clasei*) pentru crearea, deschiderea, copierea, mutarea sau ștergerea fișierelor.
- **Directory** - este o clasă care expune un set de metode statice pentru crearea, deschiderea, copierea, mutarea sau ștergerea directoarelor.
- **Path** – este o clasă care expune metode de manipulare a căilor de directoare.
- **FileInfo** – este o clasă a cărei obiecte reprezintă fișiere fizice pe disc și expune metode de manipulare a acestor fișiere. Pentru operații de scriere-citire asupra acestor fișiere va trebui creat un obiect **Stream**.
- **DirectoryInfo** - este o clasă a cărei obiecte reprezintă directoare fizice pe disc și expune metode de manipulare a acestor directoare.
- **FileStream** – un obiect al acestei clase reprezintă un fișier care poate fi scris, citit sau scris și citit.
- **StreamReader** – obiectele acestei clase sunt utilizate pentru citirea unui stream dintr-un fișier de tip **FileStream**.
- **StreamWriter** - obiectele acestei clase sunt utilizate pentru scrierea unui stream într-un fișier de tip **FileStream**

Aparent, clasele **File** și **FileInfo**, respectiv **Directory** și **DirectoryInfo** expun metode similare. Poate apare întrebarea care din clase este mai bine să fie folosite.

În realitate metodele nu sunt chiar identice, iar pentru cele care sunt, alegerea rămâne la latitudinea programatorului.

1.2 Clasele File și Directory

Așa cum am arătat mai sus, sunt clase care expun un set de metode statice pentru manipularea fișierelor și directoarelor. Prin intermediul lor, fișierele și directoarele pot fi manipulate fără a crea instanțe ale acestor clase. Câteva metode mai des utilizate ar fi:

File

- **public static FileStream Create (String path)** – crează un fișier cu numele specificat în variabila path. Returnează un obiect **FileStream**, care poate fi utilizat pentru scrierea-citirea datelor la nivelul fișierului.
- **public static FileStream Open (String path, FileMode mode)** – deschide un fișier specificat de calea path în modul descris de mode. Returnează obiectul **FileStream** pentru manipularea datelor. Enumerarea FileMode specifica modul de deschidere al fișierului. Membrii acestei enumerări sunt (fără a mai preciza semnificația lor, fiind foarte semănători cu ce ați învățat în cursul de C): Append, Create, CreateNew, Open, OpenOrCreate, Truncate.
- **public static void Delete (string path)** – șterge fișierul cu numele specificat de variabila path.
- **public static void Copy (String sourceFileName, String destFileName)** – copiază fișierul cu numele sourceFileName în fișierul cu numele destFileName. Ca o observație, acesta din urmă nu trebuie să existe, pentru că la copiere este creat automat.
- **public static void Move (String sourceFileName, String destFileName)** – mută fișierul sursă în destinație.
- **public static bool Exists (string path)** - returnează true dacă fișierul specificat există.

Directory

- **public static DirectoryInfo CreateDirectory (String path)** – crează directorul cu numele specificat de variabila path. Returnează un obiect **DirectoryInfo** care reprezintă chiar acel director. Despre clasa **DirectoryInfo** vom vorbi mai jos.
- **public static void Delete (String path)** – șterge directorul cu numele specificat de variabila path. Acest director trebuie să fie gol în momentul ștergerii.
- **public static string GetCurrentDirectory ()** – returnează calea spre directorul curent.
- **public static void SetCurrentDirectory (String path)** – setează directorul curent la calea path.
- **public static String[] GetDirectories (String path)** – returnează un șir de nume de directoare, reprezentând subdirectoarele directorului cu numele conținut în variabila path.
- **public static string[] GetFiles (String path)** – returnează numele fișierelor.

- **public static bool Exists (string path)** – returnează true dacă directorul specificat există.
- **public static void Move (String sourceDirName,String destDirName)** – mută directorul sourceDirName la calea destDirName.

Toate metodele în caz de eroare aruncă un set de excepții. Le vom detalia în momentul implementării unui exemplu.

1.3 Clasele FileInfo și DirectoryInfo

Spre deosebire de clasele anterioare, aceste clase nu expun metode statice. Metodele expuse de acestea pot fi utilizate doar prin intermediul unor obiecte ale claselor.

Un obiect **FileInfo** reprezintă un fișier fizic pe disc. El nu este un stream! Pentru manipularea datelor acestuia, va trebui creat și asociat un obiect **FileStream**. Implementarea acestui mecanism o vom descrie mai târziu.

Construcția unui obiect **FileInfo** se face prin specificarea pentru constructorul clasei a căii și numelui fișierului asociat. În exemplul de mai jos, se creează fișierul "C:/postuniv.exemplu.txt".

```
FileInfo fisier=new FileInfo("C:/postuniv.exemplu.txt");
```

Pe lângă metodele expuse, în mare parte asemănătoare cu metodele clasei **File**, clasa **FileInfo** expune și un set de proprietăți utile pentru manipularea fișierelor. Câteva din acestea sunt:

- **Attributes** – conține atributele fișierului asociat.
- **CreationTime** – conține data și ora creării fișierului asociat.
- **DirectoryName** – numele directorului în care se găsește fișierul.
- **Exists** – true dacă fișierul cu numele specificat există.
- **FullName** – numele complet (cale+nume) pentru fișier.
- **Length** – lungimea fișierului.
- **Name** – numele (fără cale) fișierului.

Clasa **DirectoryInfo** expune în mare parte aceleași metode și proprietăți ca și clasa **FileInfo**.

1.4 Obiecte FileStream

Clasa **FileStream** implementează obiecte care reprezintă fișiere pe disc. Este utilizată în general pentru a manipula adte de tip octet. Pentru manipularea datelor de tip caracter, este mai simplu să se utilizeze alte 2 clase, respectiv **StreamReader** și **StreamWriter**.

Există mai multe căi prin care se poate crea un obiect **FileStream**. Cea mai simplă metdă este de a preciza constructorului clasei numele fișierului asociat și modul de deschider a fișierului:

```
FileStream fs=new FileStream("fis.txt", FileMode.OpenOrCreate);
```

În acest caz, obiectului `fs` îi este asociat fișierul "fis.txt", care este deschis dacă există, respectiv creat dacă nu există. Enumerarea `FileMode` oferă mai multe moduri de deschidere a unui fișier:

- **Append** – deschide sau creează un fișier, fără să șteargă datele existente. Datele nou introduse se adaugă la sfârșitul fișierului. Fișierul trebuie să fie deschis în scriere.
- **Create** – creează un nou fișier gol. Dacă el deja există, datele conținute se pierd.
- **CreateNew** – creează un nou fișier. Dacă un fișier cu numele specificat există, se generează o excepție.
- **Open** – deschide un fișier existent. Dacă acesta nu există, se generează o excepție.
- **OpenOrCreate** – deschide un fișier existent, sau, dacă nu există, îl creează.

Elementele `FileMode` se pot utiliza în conjuncție cu elemente din enumerarea `FileAccess`, care precizează modul de acces la datele fișierului. Acestea sunt:

- **Read** – fișierul este deschis în citire.
- **Write** – fișierul este deschis în scriere.
- **ReadWrite** – fișierul este deschis în scriere și citire.

Deci, o altă formă de creare a unui obiect **FileStream** este

```
FileStream fs=new FileStream("fis.txt",FileMode.Open,  
FileAccess.Write);
```

De altfel, și clasele **File** și **FileInfo** expun metode de deschidere a unui fișier: `OpenRead()`, respectiv `OpenWrite()`. Deci, construcția unui obiect **FileStream** poate fi făcută și în unul din modurile de mai jos:

```
FileStream fs=File.OpenRead("fis.txt");
```

sau

```
FileInfo fi=new FileInfo("fis.txt");  
FileStream fs=fi.OpenRead();
```

1.4.1 Poziționarea într-un fișier

Metoda clasei `FileStream` care permite poziționarea în interiorul unui fișier este

```
public long Seek(long offset, SeekOrigin origin)
```

Parametrul `offset` specifică numărul de octeți peste care se face saltul, iar parametrul `origin` specifică originea saltului. Enumerarea `SeekOrigin` are următoarele elemente:

- **Begin** – saltul se face cu `offset` octeți, numărați de la începutul fișierului.

- **Current** - saltul se face cu offset octeți, numărați de la poziția curentă în fișier..
- **End** - saltul se înapoi face cu offset octeți, numărați de la sfârșitul fișierului.

Din momentul apelului metodei, noua poziție devine poziție curentă în fișier. Saltul în afara fișierului generează excepție.

Cîteva exemple de utilizare ar fi:

```
FileStream fi=File.OpenRead("fis.dat");
fi.Seek(12,SeekOrigin.Begin);
```

Poziția curentă în fișier este octetul al 12-lea, numărat de la începutul fișierului.

```
fi.Seek(10,SeekOrigin.Current);
```

Se face un salt de 10 octeți față de poziția curentă.

```
fi.Seek(-8,SeekOrigin.End);
```

Poziția curentă în fișier este cu 5 octeți înainte de sfârșitul fișierului.

Evident, orice operație de citire-scriere ulterioară se va face de la noua poziție curentă în fișier.

1.4.2 Citirea și scrierea datelor din fișier

Clasa **FileStream** furnizează și metode de scriere-citire. Le vom prezenta, dar nu vom intra în amănunte, deoarece marea majoritate a operațiilor intrare-ieșire se realizează cu ajutorul claselor **StreamReader** și **StreamWriter**.

Citirea octeților dintr-un fișier se face cu metoda

```
public int Read(byte[] array, int offset, int count)
```

Primul paramtru este numele unui șir de octeți în care se vor stoca octeții citați din fișier. Al doilea paramtru specifică poziția în șir începînd cu care se vor depune octeții, iar al treilea câți octeți vor fi citați. În caz de eroare, metoda generează excepție. De exemplu secvența:

```
byte[] data=new byte[100];
try
{
    FileStream fi=new FileStream("fis.dat",FileMode.Open);
    fi.Seek(20,SeekOrigin.Begin);
    fi.Read(data,40,10);
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

va citi 10 octeți din fișier, începând cu poziția 20 și îi va stoca în șirul `data` începând cu indexul 40.

Similar, scrierea de octeți în fișier se face cu ajutorul metodei
`public int Write(byte[] array, int offset, int count)`

Exemplul

```
byte[] data=new byte[100];
for (int i=0;i<30;i++)
    data[i]=i;
try
{
    FileStream fi=new FileStream("fis.dat",FileMode.OpenOrCreate);
    fi.Seek(0,SeekOrigin.Begin);
    fi.Write(data,0,30);
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

va scrie primii 30 de octeți din șirul `data` în fișier, începând cu poziția 0.

1.5 Clasele **StreamReader** și **StreamWriter**

Atunci când avem de transferat caracter, clasele **StreamReader** și **StreamWriter** oferă o modalitate mult mai elegantă de lucru. Clasa **StreamReader** permite citirea șirurilor de caracter dintr-un fișier, iar clasa **StreamWriter** scrierea șirurilor de caractere. Metodele acestor clase permit manipularea de obiecte **String**, cu toate avantajele care decurg din aceasta.

Modalitatea de construcție a obiectelor **StreamReader** și **StreamWriter** este foarte simplă: fie se construiește un obiect **FileStream** și noul obiect **StreamReader** sau **StreamWriter** se asociază acestuia, sau pur și simplu obiectele sunt construite prin furnizarea către constructor a numelui fișierului. Secvența

```
FileStream fi=new FileStream("fis.txt", FileMode.Create);
StreamWriter sw=new StreamWriter(fi);
sw.WriteLine(" Salut!");
sw.Write("Putem inchide");
sw.Close();
```

va crea un obiect **StreamWriter** asociat fișierului "fis.txt" nou creat și va scrie cele 2 texte, în primul caz trecând pe linie nouă după scrierea textului. O secvență similară, poate fi

```
StreamWriter sw=new StreamWriter("fis.txt",false);
sw.WriteLine(" Salut!");
sw.Write("Putem inchide");
sw.Close();
```

Efectul este același, dar constructorului clasei **StreamWriter** i se transmite direct numele fișierului și true dacă ca datele să fie adăugate în fișier, respectiv false dacă fișierul se creează gol. Deci

```
StreamWriter sw=new StreamWriter("fis.txt",false);
```

este echivalent cu

```
FileStream fi=new FileStream("fis.txt", FileMode.Create);  
StreamWriter sw=new StreamWriter(fi);
```

respectiv

```
StreamWriter sw=new StreamWriter("fis.txt",true);
```

cu

```
FileStream fi=new FileStream("fis.txt", FileMode.Append);  
StreamWriter sw=new StreamWriter(fi);
```

Obiectele **StreamReader** funcționează de o manieră similară:

```
String sir;  
FileStream fi=new FileStream("fis.txt", FileMode.Open);  
StreamReader sr=new StreamReader(fi);  
sir=sr.ReadLine();  
sr.Close();
```

va citi o linie din fișierul "fis.txt" și o va salva în stringul sir.

```
String sir, temp;  
StreamReader sr=new StreamReader("fis.txt");  
temp=sr.ReadLine();  
while ( temp!=null)  
{  
    sir+=temp;  
    temp=sr.ReadLine();  
}  
sr.Close();
```

În secvența de mai sus, se poate observa modalitatea de construcție a obiectului **StreamReader** prin transmiterea direct către constructor a numelui fișierului. Secvența citește toate liniile din fișier (când se ajunge la sfârșitul fișierului ReadLine() returnează null) și le salvează în șirul sir.

Să construim un exemplu. Pentru aceasta să creăm un nou proiect de tip **WindowsApplication**, pe care haideți să-l numim "unu". Să construim apoi machete din fig. 1.1. Prin intermediul acestei aplicații, vom realiza citirea, editarea și salvarea conținutului fișierelor cu extensia ".txt". Macheta conține un control TextBox și 3 controale Button. Pentru controlul TextBox, să modificăm proprietatea **Name** la eFis proprietatea **Multiline** la True (permițând astfel inserarea mai multor linii în control) și proprietatea **ScrollBars** la Both (am adăugat astfel în caz de necesitate bare de

scroll orizontală și verticală). De asemenea, pentru controalele Button, să modificăm proprietățile **Name** și **Text** în felul următor (de la stânga la dreapta): open respectiv Open, save respectiv Save și exit respectiv Exit.

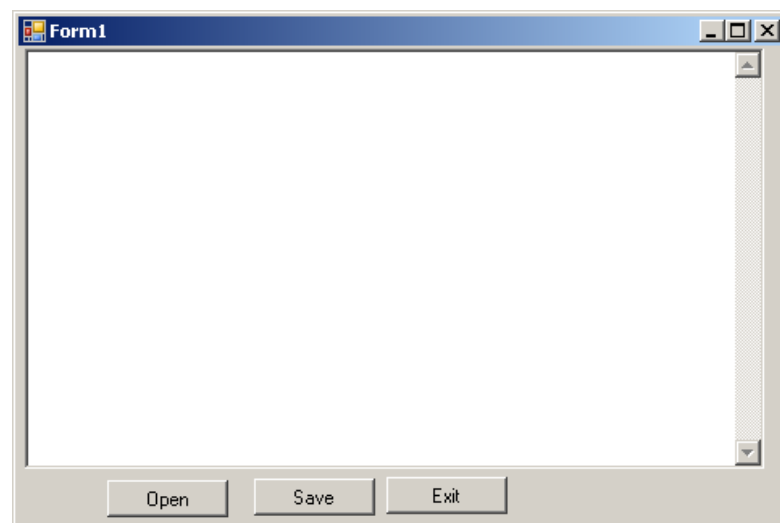


Figura 1.1

Evident, apăsarea butonului Open va permite alegerea și deschiderea unui fișier, iar butonul Save salvarea acestuia. Dorim să implementăm mecanismul de deschidere-salvare standard în sistemul de operare Windows. Pentru aceasta, infrastructura ne pune la dispoziție 2 clase, **OpenFileDialog** și respective **SaveFileDialog**, care permit manipularea fișierelor prin intermediul unor controale de dialog.

1.5.1 OpenFileDialog

Infrastructura C# oferă casete de dialog pentru manipularea numelor de fișiere. O astfel de casetă este OpenFileDialog (fig. 1.2).

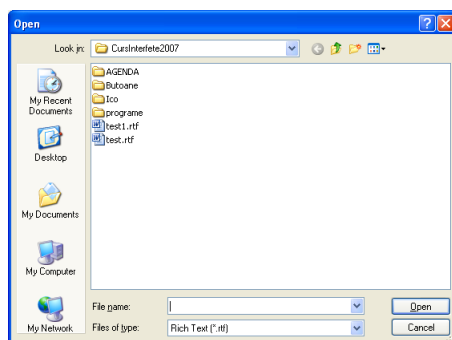


Figura 1.2

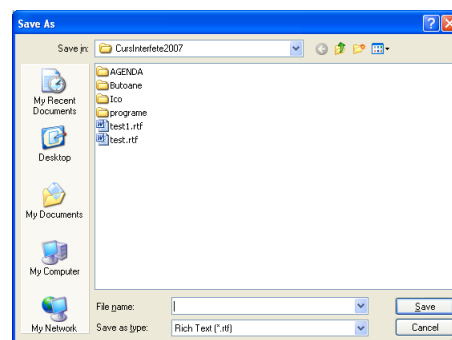


Figura 1.3

În această casetă se poate căuta numele unui fișier existent sau adăuga un nou nume. De asemenea, se permite specificarea tipului de fișier care urmează să fie încărcat. Pentru specificarea tipurilor de fișiere acceptate, trebuie construit un filtru, după următorul mecanism:

„mesaj₁ | tip₁ fișier | mesaj₂ | tip₂ fișier | ... | mesaj_n | tip_n fișier ;

Acest filtru trebuie aplicat casetei de dialog după creare cu instrucțiunea `new`, dar înainte de afișare cu funcția `ShowDialog()`. Permite utilizarea așa numitelor wildcards (semnele `?`, respectiv `*`).

Caseta de dialog oferă o serie de proprietăți utile pentru utilizator. Câteva din acestea sunt:

- **Title** – permite modificarea textului afișat în bara de titlu a casetei de dialog;
- **FileName** – numele fișierului (fișierelor) selectate;
- **InitialDirectory** – numele directorului în care începe căutarea

1.5.2 SaveFileDialog

În figura 1.3 este prezentată caseta de dialog `SaveFileDialog`. Aceasta permite salvarea cu un nume ales sau dat a unui fișier. Modul de lucru cu această casetă de dialog este absolut similar casetei `OpenFileDialog`.

Deoarece fișierele sunt entități externe programului, încercarea de accesare a unui fișier cu nume eronat poate duce la erori în cadrul programului, respectiv la oprirea lui forțată de către sistemul de operare. De aceea, înainte de a implementa funcția care se execută la apăsarea butonului `Open`, mai trebuie să vorbim despre

1.5.3 Excepții

Complexitatea aplicațiilor actuale, în special a celor orientate pe obiecte, presupune o modularizare pe nivele ierarhice. Această situație impune tratarea deosebit de riguroasă a situațiilor deosebite, în special a erorilor care pot să apară la nivelul diferitelor componente ale unui proiect.

În cele ce urmează, vom înțelege prin excepție o situație deosebită care poate să apară pe parcursul execuției unei componente soft parte a unei aplicații. Erorile pot fi privite ca și un caz particular de excepții, dar în general nu toate excepțiile sunt erori.

Excepțiile se împart în două categorii: excepții *sincrone* și excepții *asincrone*. Excepțiile sincrone sunt excepții a căror apariție poate fi prevăzută de programator (de exemplu, un fișier care se dorește a fi prelucrat nu se află în directorul așteptat). Excepțiile asincrone sunt excepții a căror apariție nu poate fi prevăzută în momentul implementării programului (de exemplu, defectarea accidentală a hard-disk-ului). Mecanismul de tratare a excepțiilor ia în considerare numai situația excepțiilor sincrone.

Problema se pune astfel: în faza de programare, se poate presupune că un modul de program va detecta în anumite situații o excepție, dar nu va putea oferi o soluție generală de tratare a acesteia, în timp ce un alt modul al aplicației poate oferi o soluție de tratare a excepției, dar nu poate detecta singur apariția excepțiilor.

Exemple tipice de excepții: împărțire cu 0, accesul la un fișier inexistent, operații cu o unitate logică nefuncțională, etc.

Uzual, la apariția unei excepții, sistemul de operare va opri forțat execuția modului de program în care excepția s-a produs, pentru ca acesta să nu afecteze buna funcționare a celorlalte programe. De multe ori însă este necesar ca excepția apărută să nu fie transmisă sistemului de operare, ci să fie "*prinsă*" și tratată în interiorul modulului care a generat-o. Acest lucru este posibil prin intermediul unui mecanism bazat pe 2 instrucțiuni: `try` și `catch`.

Mecanismul este următorul: zona critică, adică zona de cod în care programatorul prevede posibilitatea apariției unei excepții este scrisă într-un bloc try. Blocul try este urmat de unul sau mai multe blocuri catch, câte unul pentru fiecare tip de excepție ce poate să apară. Execuția programelor în acest caz este prezentată în fig. 1.4 a și b.

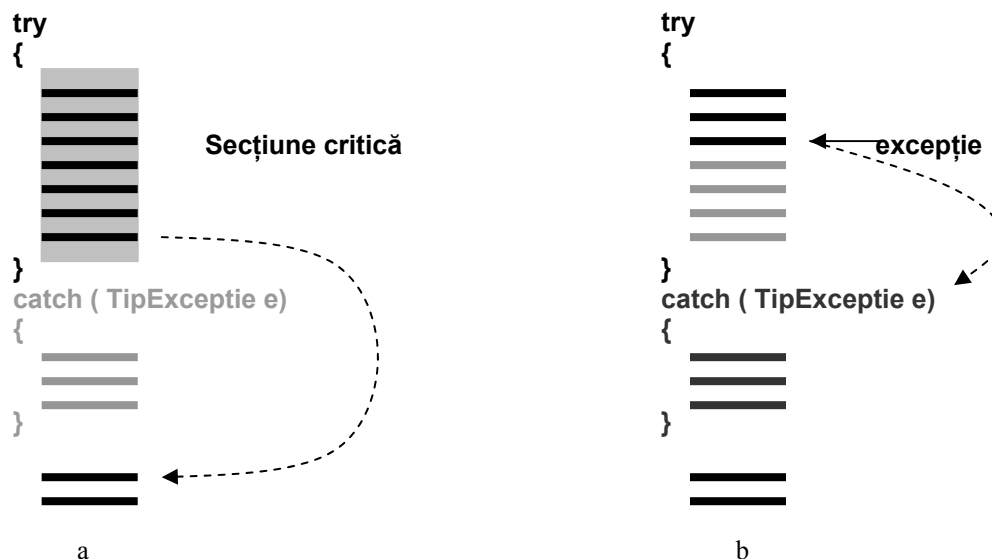


Figura 1.4

Fig. 1.4a prezintă situația normală de execuție, când în secțiunea critică nu apare excepție. În acest caz, se execută în succesiune normală toate instrucțiunile din secțiunea critică, după care, următoarea instrucțiune executată este prima de după blocul catch, acesta fiind neglijat.

Fig. 1.4b prezintă situația apariției unei excepții. În acest caz, următoarea instrucțiune executată după cea care a generat excepția este prima din blocul catch, în acesta urmând a fi tratată excepția. După execuția instrucțiunilor din blocul catch, se continuă în secvență normală, cu prima instrucțiune de după acesta.

Marea majoritate a claselor care lucrează cu entități externe memoriei generează excepții, pentru ca posibilele erori să poată fi tratate încă din faza de programare. Este și cazul claselor care lucrează cu fișiere și este indicat ca toate operațiile făcute asupra fișierelor să fie tratate prin mecanismul try-catch.

Și acum să trecem la implementare. Vom apăsa dublu-click pe butonul Open și vom implementa codul:

```
private void open_Click(object sender, EventArgs e)
{
    eFis.Clear();
    OpenFileDialog op = new OpenFileDialog();
    op.InitialDirectory = Directory.GetCurrentDirectory();
    op.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
    if (op.ShowDialog() == DialogResult.OK)
    {
        String strLin;
        try
        {
            StreamReader sr = new StreamReader(op.FileName);
            strLin = sr.ReadLine();
            while (strLin != null)
            {
```

```

        eFis.Text += strLin + "\r\n";
        strLin = sr.ReadLine();
    }
    sr.Close();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
}
}

```

Conținutul funcției este foarte ușor de înțeles dacă ținem cont de cele prezentate anterior. Se șterge conținutul controlului TextBox și se crează un obiect OpenFileDialog, pentru care se definește filtrul și directorul inițial de căutare ca fiind directorul curent. Apoi se afișează controlul. La revenire din caseta OpenFileDialog, dacă s-a revenit apăsând butonul OK, se creează un StreamReader asociat fișierului ales în caseta OpenFileDialog (prin transmiterea către StreamReader a numelui acestuia) și conținutul fișierului este citit linie cu linie. Fiecare linie citită este adăugată proprietății **Text** a controlului TextBox, după care se adaugă șuril CR-LF.

De o manieră similară se implementează și funcția executată la apăsarea butonului Save.

```

private void save_Click(object sender, EventArgs e)
{
    SaveFileDialog sv = new SaveFileDialog();
    sv.InitialDirectory = Directory.GetCurrentDirectory();
    sv.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
    if (sv.ShowDialog() == DialogResult.OK)
    {
        try
        {
            StreamWriter sr = new StreamWriter(sv.FileName, false);
            sr.WriteLine(eFis.Text);
            sr.Close();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
}

```

Această funcție nu mai necesită explicații. Și în fine, ultima implementare,

```

private void exit_Click(object sender, EventArgs e)
{
    Application.Exit();
}

```

Compilați și executați programul.

1.6 Serializarea datelor

Până acum am văzut cum se pot salva și restaura în și din fișiere datele de tip octet sau caracter. Dar ce e de făcut, dacă avem de salvat și restaurat date mai complexe, de exemplu obiecte ale unei clase? Evident, obiectele se pot salva sub formă de șiruri de octeți, dar va fi foarte greu la restaurare să împărțim octeții astfel încât să recompunem obiectele salvate. Din fericire, majoritatea limbajelor de nivel înalt, inclusiv C#, implementează un mecanism simplu și fiabil de salvare și restaurare a obiectelor prin intermediul fișierelor. Acest mecanism poartă denumirea de *serializare*.

Orice clasă ale cărei obiecte pot fi salvate sau restaurate prin intermediul fișierelor, va trebui fie să fie declarată serializabilă, fie să implementeze o interfață specifică, numită **ISerializable**. În momentul în care infrastructura primește o cerere de serializare a unor obiecte, va verifica întâi dacă clasa din care fac parte obiectele implementează interfața **ISerializable** și dacă nu, va verifica dacă a fost declarată ca fiind serializabilă. Dacă nici una din condiții nu este îndeplinită, obiectele nu vor putea fi serializate.

Declararea unei clase ca fiind serializabilă este foarte simplă. Pur și simplu, clasei îi este specificat atributul **[Serializable]**. Să lămurim printr-un exemplu. Să creăm un nou proiect, fie unu-unu numele acestuia și proiectului să-i adăugăm o nouă clasă, numită **Rezervare**: click dreapta pe rădăcina unu-unu în Solution explorer, Add, Class..., și la Name scriem Rezervare. Să adăugăm attributele clasei, un constructor și s-o declarăm serializabilă.

```
namespace unu_unu
{
    [Serializable]
    class Rezervare
    {
        public Rezervare()
        {
        }
        public String Nume;
        public DateTime DataS;
        public DateTime DataP;
        public int nrCam;
    }
}
```

Am adăugat clasei un constructor, 4 attribute de tip String, DateTime și int. Obiectele clasei pot reprezenta de exemplu rezervări la un hotel, pentru care se specifică numele clientului, data sosirii, data plecării și numărul de camere rezervate. Prin adăugarea atributului **[Serializable]**, clasa a fost declarată automat serializabilă, deci instanțele ei vor putea fi salvate și restaurate din fișiere.

Să implementăm acum interfața de lucru. Vom implementa interfața din

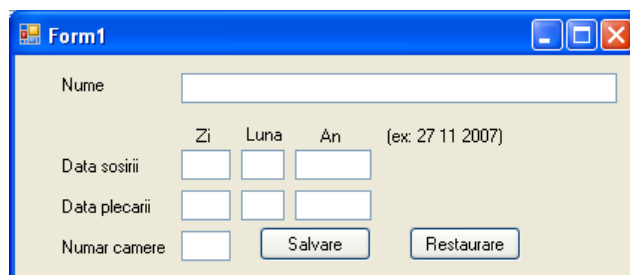


Figura 1.5

figura 1.5, pentru care controalele TextBox au următoarele attribute Name (de sus în jos și de la stânga la dreapta): nume, zis, ls, ans, zip, lp, anp, respectiv nrc, iar butoanele (de la stânga la dreapta) salvare și restaurare.

Transformarea obiectelor în șiruri de octeți pentru a fi executat transferul la nivelul fișierului, se face utilizând *formatori* (Formatters). Există 2 tipuri de formatori: BinaryFormatter și SoapFormatter. Pe primul îl vom folosi aici, pentru a formata obiectele în și din șiruri de octeți, iar pe al doilea îl vom întâlni în capitolele viitoare. Pentru a putea utiliza aceste formatoare, va trebui să adăugăm fișierului care implementează forma spațiul de nume System.Runtime.Serialization.Formatters.Binary, iar pentru a putea implementa mecanismul efectiv de serializare, spațiul de nume System.Runtime.Serialization. Să ne reamănim de asemenea, că pentru a putea lucra cu fișiere, va trebui să adăugăm și spațiul de nume System.IO:

```
using System;
using System.Collections.Generic;
...
using System.Windows.Forms;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
```

```
namespace unu_unu
{
    public partial class Form1 : Form
    {
        ...
    }
}
```

Vom adăuga clasei **Form1** un obiect **Rezervare**, care să fie vizibil în toate metodele clasei:

```
public partial class Form1 : Form
{
    Rezervare rez = new Rezervare();
    public Form1()
    {
        ...
    }
}
```

Și acum să implementăm metodele de srializare și deserializare a obiectelor **Rezervare**. Vom adăuga clasei o metodă care realizează serializarea, numită Salvare().

```
public void Salvare()
{
    Stream s = File.Open("rezervare.rez", FileMode.Create);
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(s, rez);
    s.Close();
    MessageBox.Show("Salvat");
    this.Close();
}
```

Cum lucrează metoda? Întâi creaza un Stream asociat fișierului “rezervare.rez” pe care-l deschide în mod creare. Crează apoi un nou obiect de tip BinaryFormatter, prin intermediul căruia apelează metoda

```
public void Serialize (Stream serializationStream, Object graph)
```

unde *serializationStream* este stream-ul în care se serializează obiectul *graph*. Cu alte cuvinte, în cazul nostru, obiectul rez de clasă **Rezervare** va fi salva în streamul *s*, adică în fișierul “rezervare.rez”. Gata. E simplu, formatorul s-a ocupat singur de serializarea obiectului.

Să vedem cum arată restaurarea. Vom implementa metoda Refacere():

```
public void Refacere()  
{  
    Stream s = File.Open("rezervare.rez", FileMode.Open);  
    BinaryFormatter bf = new BinaryFormatter();  
    Object obj = bf.Deserialize(s);  
    rez = obj as Rezervare;  
}
```

Simplu din nou. Pentru deserializare am folosit metoda (care cred că nu mai necesită explicații suplimentare)

```
public Object Deserialize (Stream serializationStream)
```

Deoarece metoda salvează obiectul serializat într-un obiect generic de clasă **Object**, pe acesta l-am particularizat ulterior la clasa **Rezervare**. Evident, metoda se poate rescrie cu ajutorul conversiei explicite de tip (cast):

```
public void Refacere()  
{  
    Stream s = File.Open("rezervare.rez", FileMode.Open);  
    BinaryFormatter bf = new BinaryFormatter();  
    rez = (Rezervare)bf.Deserialize(s);  
}
```

Ambele modalități de scriere sunt echivalente. Acum tot ce ne mai rămâne de făcut e să implementăm funcțiile care răspund la apăsarea butoanelor:

```
private void salvare_Click(object sender, EventArgs e)  
{  
    rez.Nume = nume.Text;  
    rez.DataS = Convert.ToDateTime(zis.Text + "." + ls.Text + "." +  
                                   ans.Text);  
    rez.DataP = Convert.ToDateTime(zip.Text + "." + lp.Text + "." +  
                                   anp.Text);  
    rez.nrCam = Convert.ToInt16(nrc.Text);  
    Salvare();  
}
```

Nu am făcut altceva decât să completăm attributele obiectului rez cu datele din controalele TextBox și să convertim mărimile asociate la tipul DateTime. După care am apelat funcția de serializare.

```

private void restaurare_Click(object sender, EventArgs e)
{
    Refacere();
    if (rez != null)
    {
        nume.Text = rez.Nume;
        String x;
        int i1, i2;
        x = Convert.ToString(rez.DataS);
        i1 = x.IndexOf('.');
        i2 = x.LastIndexOf('.');
        zis.Text = x.Substring(0, i1);
        ls.Text = x.Substring(i1 + 1, i2-i1-1);
        ans.Text = x.Substring(i2+1, 4);
        x = Convert.ToString(rez.DataP);
        i1 = x.IndexOf('.');
        i2 = x.LastIndexOf('.');
        zip.Text = x.Substring(0, i1);
        lp.Text = x.Substring(i1 + 1, i2 - i1 - 1);
        anp.Text = x.Substring(i2 + 1, 4);
        nrc.Text = Convert.ToString(rez.nrCam);
    }
    else
        MessageBox.Show("Citire nereusita");
}

```

Ce am făcut? Am deserializat din fișier obiectul rez, după care componentele lui sunt afișate în controalele TextBox. Deoarece stringul rezultat în urma conversiei din DateTime este de forma "zz.ll.aaa", am extras subșirurile corespunzătoare zilei, lunii și anului, pentru a putea fi afișate. Dacă deserializarea a reușit, obiectul rez se încarcă cu datele preluate din fișier. În caz contrar, funcția Deserialize() returnează null.

Compilați și executați programul.