Depanarea si tratarea erorilor

In programare sunt posibile doua tipuri de erori: *erori fatale* si *erori semantice* (sau *logice*). Erorile fatale includ erori simple care impiedica compilarea aplicatiilor (erori de sintaxa) sau probleme mai serioase care apar la executia programelor (exceptii). Erorile semantice sunt mult mai subtile. Spre exemplu, aplicatia creata esueaza in a inregistra un set de date intr-o baza de date deoarece un anumit camp lipseste, sau inregistreaza setul de date in mod eronat. Erorile de acest tip demonstreaza ca logica programului este defectuasa. Ele se mai numesc si erori logice.

Daca erorile fatale sunt relativ usor de detectat si tratat, de multe ori nici nu stim de existenta vreo unei erori logice pana cand vreun utilizator al aplicatiei noastre reclama ca ceva nu functioneaza cum ar trebui. Avem astfel sarcina de a urmari codul programului si de a afla ce nu functioneaza in mod corect. In astfel de situatii, tehnicile de depanare puse la dispozitie atat de limbajul C# cat si de Visual Studio (VS) sunt de mare ajutor.

Procesul prin care sunt identificate si corectate acele portiuni de cod care nu functioneaza asa cum s-a intentionat initial se numeste *depanare (debugging).*

Asa cum ati observat, puteti executa o aplicatie in doua moduri: cu depanarea activata (Start Debugging sau F5) si cu depanarea dezactivata (Start Without Debugging sau Ctrl+F5).

De asemenea, VS permite construirea *aplicatiilor in doua configuratii: Debug si Release*.

Atunci cand este construita o aplicatie in configuratia Debug, nu este executat doar codul programului ci se intampla si altceva. Blocurile de depanare (debug builds) mentin o informare simbolica privind programul nostru astfel ca VS stie ce se intampla cu fiecare linie de cod executata. Aceaste informatii sunt continute in fisierele .pdb din directoarele Debug. Acest aspect ne permite realizarea urmatoarelor operatii: *transmiterea de informatii catre VS, urmarirea si editarea valorilor variabilelor in timpul executiei aplicatiei, oprirea executiei la anumite puncte marcate in codul programului, executia programului linie cu linie, monitorizarea unei variabile in timpul executiei*, etc.

In configuratia Release, codul aplicatiei este optimizat, iar operatiile enumerate mai sus nu sunt posibile. Blocurile release (Release builds) se executa mai rapid, astfel cand realizati o aplicatie, de regula veti furniza utilizatorilor blocuri release intrucat informatia simbolica pe care blocurile debug le contin nu este necesara.

In cele ce urmeaza vom expune cateva tehnici de depanare. Acestea sunt grupate in doua categorii, dupa modul in care acestea sunt utilizate. Asfel, depanarea se poate realiza fie prin intreruperea executiei programului (modul break) in diverse puncte ale acestuia, fie prin consemnarea unor observatii care sa fie analizate ulterior.

Depanare in modul normal (nonbreak)

Sa consideram urmatoarea secventa de cod:

```
Console.WriteLine("Functia F() urmeaza a fi apelata");
F(); //Functia F() realizeaza o serie de operatii
Console.WriteLine("Functia F() a fost apelata");
```

Codul de mai sus arata cum putem obtine mai multe informatii privind functia F(). Acest aspect este important in multe situatii, insa conduce la o dezordine in ceea ce priveste informatiile afisate in consola. Mai mult, in cazul aplicatiilor Windows nici nu utilizam consola. O cale alternativa este de a afisa astfel de informatii intr-o fereastra separata, fereastra *Output* care se poate selecta din meniul View. Aceasta fereastra ofera informatii legate de compilarea si executia codului, erorile intalnite in timpul compilarii sau poate fi utilizata pentru a afisa informatii de genul celor prezentate mai sus, utile in depanare.

O alta cale este de a crea un fisier separat (*logging file*) care sa adauge diverse informatii legate de depanare atunci cand aplicatia este executata.

Pentru a scrie text in fereastra *Output* avem doua posibilitati. Fie utilizam metodele puse la dispozitie de clasele Debug sau Trace care fac parte din spatiul de nume System.Diagnostics, fie utilizam o facilitate oferita de Visual Studio, si nu de limbajul C#, si anume anumite "marcaje" in program numite *tracepoints.*

Sa consideram mai intai prima posibilitate. La fel ca in cazul clasei Console, cele doua clase amintite mai sus, pun la dispozitie metodele WriteLine() si Write(), insa si metode precum WriteLineIf(), WriteIf() care contin in plus un parametru de tipul bool. Acestea doua din urma scriu text doar daca *parametrul bolean este true.*

Metodele din clasele Debug si Trace opereaza in acelasi mod, insa cu o diferenta importanta. Metodele din *clasa Debug functioneaza doar cu blocurile debug*, in timp ce metodele din *clasa Trace functioneaza cu oricare din blocuri (release sau debug)*. Astfel, metoda Debug.WriteLine() nu se va compila intr-o versiune release a aplicatiei, in timp ce metoda Trace.WriteLine() se compileaza atat in versiunea release cat si versiunea debug.

Metodele Debug.WriteLine() , Trace.WriteLine() nu functioneaza la fel ca metoda Console.WriteLine(). Acestea opereaza doar asupra unui parametru de tip string, si nu permit sintaxe de genul {0} sau {0:#.##}. Optional, se poate utiliza un al doilea parametru de tip string care afiseaza o categorie pentru textul ce urmeaza a fi afisat in fereastra Output. Forma generala a mesajului afisat este:

<category>: <message>

Spre exemplu, urmatoarea linie de cod:

Debug.WriteLine("Aduna i cu j","FunctiaSuma")

afiseaza:

FunctiaSuma: Aduna i cu j

Pentru a beneficia de optiunile de afisare de care de bucura metoda Console.WriteLine(), se poate utiliza metoda Format() a clasei string (vezi exemplul urmator).

Construiti urmatoarea aplicatie atat in configuratia debug cat si in configuratia release. Pentru ca aceasta sa poata fi compilata, includeti o referinta catre asamblajul System.

```
using System;    using System.Diagnostics;
using System.Collections;
namespace ConsoleApplication5
{ class Program
  {
   static int ValoareMaxima(int[] intArray, out ArrayList listaIndecsi)
    {     Debug.WriteLine("Cautarea valorii maxime a inceput");
          int valMax = intArray[0];
          Debug.WriteLine(string.Format("Valoarea maxima a fost
initializata cu {0} la elementul cu indexul 0",valMax));
          for (int i = 1; i < intArray.Length; i++)
          {  Debug.WriteLine(string.Format("Acum testam elementul
cu indexul {0} ", i));
             if (intArray[i] > valMax)
             {  valMax = intArray[i];
                Debug.WriteLine(string.Format("Un nou maxim gasit.
Noua valoare este {0} la elementul cu indexul {1}",valMax,i));
             }
          }
          Trace.WriteLine(string.Format("Valoarea maxima din sir
este: {0} ",valMax));

         Debug.WriteLine(string.Format("Acum testam care sunt
           elementele din sir care au valoarea maxima"));
         listaIndecsi = new ArrayList();
        for (int i = 0; i < intArray.Length; i++)
         {  Debug.WriteLine(string.Format("Acum testam elementul
          cu indexul {0} ", i));
          if (intArray[i] == valMax)
             {  valMax = intArray[i];
                listaIndecsi.Add(i);
                Debug.WriteLine(string.Format("Un nou element
           gasit care are valoarea {0}. Elementul cu indexul {1}",
           valMax,  i));
             }
         }
         Trace.WriteLine(string.Format("Elementele din sir care au
           valoarea maxima au indecsii:"));
         foreach(int index in listaIndecsi)
         {      Trace.Write(index.ToString()+"\t");        }
         Trace.WriteLine("");
         return valMax;
    }
    static void Main(string[] args)
    {  int[] myArray = { 1, 8, 3, 9, 6, 2, 5, 9, 3, 9, 0, 2 };
       ArrayList valMaxIndecsi=new ArrayList();
       int maxVal = ValoareMaxima(myArray, out
         valMaxIndecsi);
       Console.WriteLine("Valoarea maxima in sir este: {0}",
         maxVal);
       Console.WriteLine("Elementele din sir care au valoarea
         maxima au indecsii:");
       foreach (int index in valMaxIndecsi)
       {  Console.Write(index.ToString() + "\t");   }
       Console.WriteLine("");
       Console.ReadKey();
    }
  }
}
```

Tracepoints. Asa cum s-a mentionat mai sus, putem scrie informatii in fereastra Output utilizand o serie de marcaje (tracepoints), facilitate pusa la dispozitie de VS.

Pentru a insera un marcaj tracepoint in codul programului, pozitionati cursorul pe linia dorita (marcajul facut va fi procesat inainte ca linia de cod sa fie executata), click dreapta si selectati Breakpoint ->Insert Tracepoint. In fereastra de dialog care apare pe ecran inserati stringul dorit. Daca doriti sa afisati valoarea unei variabile atunci includeti numele variabilei in acolade. De asemenea, puteti obtine si alte informatii importante utilizand cuvinte cheie precum $FUNCTION, $CALLER, etc. sau este posibil sa executati un macro. O alta facilitate este aceea de a intrerupe executia programului (deci sa actioneze ca un breakpoint). Prin apasarea butonului OK, un romb rosu va aparea in stanga liniei de cod.

Executand un program care are inserate marcaje tracepoints, vom obtine aceleasi rezultat ca atunci cand utilizam clasele Debug si Trace in modul Debug (vezi exemplul precedent). Puteti sterge un astfel de marcaj sau sa-l dezactivati prin click dreapta si Disable Breakpoint.

Principalul avantaj pe care il ofera utilizarea marcajelor este rapiditatea si usurinta cu care sunt adaugate aplicatiei. Drept bonus este obtinerea de informatii suplimentare precum $FUNCTION (numele functiei) $PID (id-ul procesului) etc.

Dezavantajul principal este ca nu avem echivalent pentru comenzi care apartin clasei Trace, asadar nu se pot obtine informatii atunci cand aplicatia utilizeaza blocuri release.

In practica se procedeaza astfel: se utilizeaza spatiul de nume Diagnostics atunci cand se doreste ca intodeauna sa se obtina informatii suplimentare despre program, in particular cand stringul de obtinut este complex si implica multe informatii despre variabile. In plus, sunt posibile doar comenzile clasei Trace in configratia Release. Marcajele tracepoints sunt utilizate pentru o analiza rapida a codului in scopul corectarii erorilor semantice.

Depanare in modul break

Facilitatile de depanare oferite de modul break pot fi utilizate prin intermediul toolbarul Debug (care poate fi inclus in toolbar-ul VS prin click dreapta pe toolbar-ul VS, iar apoi selectati Debug) fereastra Breakpoints (meniul Debug -> Windows -> Breakpoints) si o serie de alte ferestre care devin active odata ce s-a intrat in modul break.

Primele patru butoane ale toolbar-ului Debug (Start, Pause, Stop, Restart) permit controlul manual al depanarii in modul break. In afara primului, celelalte sunt inactive pentru un program care nu se executa la momentul respectiv. Odata ce aplicatia este pornita (se apasa butonul Start), ea poate fi intrerupta (butonul Pause), intrandu-se astfel in modul break, sau poate fi parasita complet (butonul Stop) sau restartata (butonul Restart).

Intreruperea aplicatiei cu butonul Pause este unul dintre cele mai simple moduri de a intra in modul break. Insa acesta nu ne permite controlul asupra locului unde aplicatia sa se intrerupa. In acest sens, se utilizeaza breakpoint-urile.

Breakpoints. Un breakpoint reprezinta un punct (marcaj) in codul sursa care declanseaza intrarea automata in modul break.

Un breakpoint poate fi configurat pentru a face urmatoarele: intra in modul break imediat ce breakpoint-ul este atins, intra in modul break daca o expresie boleana este evaluata cu true, intra in modul break daca un breakpoint este atins de un anumit numar de ori, intra in modul break odata ce breakpoint-ul este atins, iar o anumita variabila si-a schinbat valoarea fata de ultima data cand breakpoint-ul a fost atins, afiseaza un text in fereastra Output sau executa un macro.

Toate acestea sunt permise doar in configuratia debug. Daca aplicatia este compilata in configuratia release, toate breakpoint-urile sunt ignorate.

Exista mai multe moduri de a introduce un breakpoint, si anume: urmand aceeasi pasi ca in introducerea unui tracepoint, daca fereastra Breakpoints este activa atunci un simplu click in stanga codului, sau apasati F9.

Utilizand **fereastra Breakpoints** puteti realiza urmatoarele operatii: dezactiva/activa un breakpoint, sterge un breakpoint, edita proprietatile unui breakpoint. Coloanele Condition si Hit Count, afisate de fereastra Breakpoints, sunt cele mai importante. Acestea pot fi editate (click dreapta) pentru a oferi functionalitatea dorita fiecarui breakpoint. Spre exemplu, putem configura un breakpoint in asa fel incat sa intram in modul break daca valMax este mai mare ca 6 prin tastarea expresiei "valMax>6" si selectarea optiunii Is true (coloana Condition).

Metodele Debug.Assert() si Trace.Assert(). Un alt mod de a intra in modul break este de a utiliza una din aceste doua metode. Aceste comenzi pot intrerupe executia programului cu un mesaj precizat de programator. Ele sunt utilizate pentru a testa daca in timpul dezvoltarii aplicatiei lucrurile decurg normal si nu sunt erori semantice. Spre exemplu, pe durata unei aplicatii, o anumita variabila trebuie sa ramana mai mica decat 8. Puteti utiliza metoda Assert() pentru a confirma acest lucru. In acest sens, metoda amintita primeste un parametru de tip bolean. Atata timp cat *valoarea parametrului este true, aplicatia se executa normal*. Daca parametrul isi schimba valoarea in false atunci apare o fereasta care contine, pe langa o serie de informatii privind adresa, functia, etc. si un mesaj (un string sau doua) care reprezinta un alt parametru al metodei Assert(). Avem posibilitatea de a termina aplicatia (Abort), de a intra in modul break (Retry) sau de a continua rularea normala a aplicatiei (Ignore).

Odata ce am intrat in modul break, avem la dispozitie diverse tehnici care permit o analiza detaliata a codului si starii aplicatiei la momentul cand executia a fost intrerupta.

Monitorizarea valoriilor variabilelor Monitorizarea continutului unei variabile este un exemplu simplu care reda utilitatea depanarii in VS. Unul din cele mai simple moduri de a obtine valoarea unei variabile este de duce mouse-ul peste numele acelei variabile. In acelasi mod putem obtine valoarea unei expresii sau valorile elementelor unui vector.

Odata ce aplicatia a intrat in modul break sunt disponibile mai multe ferestre (meniul Debug -> Windows). Ferestrele: **Autos** (contine variabilele in uz si variabilele din precedentele declaratii), **Locals** (toate variabilele din acel domeniu de vizibilitate (sau bloc)), **Watch n**: (variabile si expresii care pot fi incluse la optiunea utilizatorului; n ia valori de la 1 la 4, asadar sunt posibile 4 ferestre).

Fiecare din aceste ferestre se comporta oarecum la fel, cu cateva proprietati particulare, depinde de functia specifica pe care o indeplineste. In general, fiecare fereastra contine o lista de variabile care prezinta informatii despre fiecare variabila:nume, valoare, tip. Variabilele mai complexe, precum tablourile pot fi examinate prin simbolurile + sau -.

Puteti edita continutul unei variabile prin inserarea noii valori in coloana Value. Astfel, se neglijeaza valoarea atribuita variabilei intr-un cod anterior. Aceasta operatiune merita facuta in cazul in care doriti sa incercati diverse scenarii care altfel ar necesita modificarea codului.

Fereastra Watch permite monitorizarea unor variabile la optiunea utilizatorului. Pentru a utiliza aceasta fereastra, scrieti numele variabilei sau expresiei in aceasta fereastra. Un aspect interesant, legat de aceasta fereastra, este faptul ca ea arata care dintre variabile si-a schimbat valoarea intre doua breakpoint-uri (sau de la o secventa de cod la alta daca se utilizeaza unul din butoanele Step Into, Step Over sau Step Out). Noua valoare este scrisa cu rosu.

Executia aplicatiei pas cu pas. Pana in momentul de fata am discutat despre ce se intampla in momentul in care se intra in modul break. Este momentul sa vedem cum putem executa codul programului comanda cu comanda (sau bloc cu bloc), fara a iesi din modul break.

Atunci cand se intra in modul break, apare un cursor in partea stanga a codului care urmeaza a fi executat. La acest moment, putem executa codul programului linie cu linie. In acest sens, utilizam, dupa caz, butonul al saselea (Step Into), al saptelea (Step Over) sau al optulea (Step Out) din toolbarul Debug. Astfel, Step Into executa o comanda si muta cursorul la urmatoarea comanda, Step Over similar butonului anterior, insa nu intra in blocurile interioare (inclusiv blocurile functiilor), Step Out ruleaza pana la sfarsitul blocului si reintra in modul break la instructiunea care urmeaza.

Daca executam aplicatia utilzand aceste butoane, vom observa in ferestrele descrise mai sus cum valorile variabilelor se modificari. Putem sa monitorizam astfel o variabila sau un set de variabile. Pentru codul care prezinta erori semantice, aceasta tehnica poate fi cea mai utila si eficienta.

Fereastra Immediate. Aceasta ferestra (aflata in meniul Debug Windows) permite executia unei comenzi in timp ce aplicatia ruleaza. Cel mai simplu mod de a utiliza aceasta fereastra este de a evalua o expresie. In acest sens, tastati expresia si apasati enter. Rezultatul va fi scris pe linia urmatoare.

Fereastra Call Stack. Aceasta fereastra arata modul in care s-a ajuns in locatia curenta. Adica este aratata metoda curenta f(), metoda g() care a apelat metoda curenta, metoda care a apelat metoda g() si asa mai departe. Aceasta fereasta este utila atunci cand se detecteaza o eroare intrucat putem vedea ce se intampla imediat inaintea erorii. Intrucat multe din erori au loc in blocul unor functii, aceasta fereastra este utila in gasirea sursei erorii.

Tratarea exceptiilor

Exceptii. Clase care reprezinta exceptii

O *exceptie* este o eroare care se produce la momentul executiei. O linie de cod poate sa nu se execute corect din diverse motive: depasire aritmetica, memorie insuficienta, indici in afara intervalului, fisierul din care se citesc sau se scriu datele nu poate fi deschis, etc.

Exemplul de mai jos genereaza exceptie deoarece se incearca impartirea prin zero:

```
class test {
public static void Main()
{   int a = 2, b=0;
    System.Console.WriteLine(a / b);  }  }
```

O aplicatie care si-ar propune sa verifice toate (sau aproape toate) posibilitatile ce pot aparea in executarea unei linii de cod si-ar pierde din claritate, ar fi foarte greu de intretinut, iar mare parte din timp s-ar consuma cu aceste verificari. Realizarea unei astfel de aplicatii este aproape imposibila.

Utilizand sistemul de tratare a exceptiilor din C#, nu este nevoie sa scriem cod pentru a detecta toate aceste posibile caderi. In plus codul se executa mai rapid.

Dintre avantajele introducerii exceptiilor amintim :

- abilitatea de a mentine cod clar si de a intelege logica aplicatiei mai usor;
- posibilitatea de a detecta si localiza bug-uri in cod;

In C# exceptiile sunt reprezentate prin clase. Toate clasele care prezinta exceptii trebuie sa derive din clasa predefinita Exception, care face parte din spatiul de nume System. Din clasa Exception deriva mai multe clase, doua dintre acestea, SystemException si ApplicationException implementeaza doua categorii generale de exceptii definite din limbajul C#: cele generate de motorul de executie si cele generate de programele de aplicatie. De exemplu, codul de mai sus produce o exceptie de tipul DivideByZeroException. Clasa DivideByZeroException deriva din ArithmeticException care la randul ei deriva din clasa SystemException.

Utilizatorul poate defini propriile sale clase care reprezinta exceptii, derivandu-le din clasa ApplicationException.

Notiuni de baza despre tratarea exceptiilor

Tratarea exceptiilor in C# se realizeaza utilizand patru cuvinte cheie: try, catch, throw si finally.

Instructiunile din program care trebuie verificate pentru aparitia exceptiilor sunt incluse in corpul blocului try. Daca pe parcursul executiei unui bloc try se produce o exceptie, atunci aceasta este lansata. Programul poate intercepta exceptia utilizand catch, tratand-o in conformitate cu logica situatiei. Exceptiile de sistem sunt lansate in mod automat de catre motorul de executie din C#. Pentru a lansa manual o exceptie, puteti utiliza cuvantul cheie throw. Codul care trebuie neaparat executat la iesirea dintr-un bloc try trebuie inclus intr-un bloc finally.

La baza tratarii exceptiilor stau constructiile try si catch. Acestea lucreaza in pereche (nu este posibil sa apara try fara catch sau invers). Forma generala a blocurilor try si catch este cea de mai jos:

```
try { // bloc de cod monitorizat pentru detectarea erorilor la executie}
catch(TipExceptie1 exOb){ //rutina de tratare pentru TipExceptie1}
catch(TipExceptie2 exOb){ //rutina de tratare pentru TipExceptie2} …..
catch(TipExceptieN exOb){ //rutina de tratare pentru TipExceptieN}
```

unde TipExceptie reprezinta tipul exceptiei care s-a produs, iar exOb memoreaza valoarea exceptiei (refera obiectul instanta a clasei TipExceptie). Specificarea exOb este optionala. Daca rutina de tratare nu are nevoie de acces la obiectul care reprezinta exceptia, nu este nevoie sa specificam exOb. Mai mult, este optionala specificarea intregii paranteze (TipExceptie exOb). In acest caz, catch are forma: catch {//rutina de tratare a exceptiei}.

Secventa de cod de mai sus functioneaza astfel: Se monitorizeaza codul din interiorul blocului try pentru a detecta posibilele exceptii. Daca o exceptie s-a produs, atunci aceasta este lansata si interceptata de instructiunea catch corespunzatoare. Corpul instructiunii catch care a interceptat exceptia contine secvente de cod pentru tratarea acesteia. O instructiune catch care specifica o exceptie de tip TipExceptie, intercepteaza o exceptie doar daca aceasta reprezinta un obiect instanta a clasei TipExceptie sau un obiect al unei clase derivate din TipExceptie. Spre exemplu, catch(Exception) intercepteaza toate exceptiile deoarece clasa Exception reprezinta clasa de baza pentru toate celelalte clase care reprezinta exceptii. Daca o exceptie este interceptata de o instructiune catch atunci toate celelalte instructiuni catch sunt ignorate. Executia programului continua cu instructiunea care urmeaza dupa catch.

Daca blocul try s-a executat normal atunci toate instructiunile catch sunt ignorate. Programul continua cu prima instructiune dupa ultimul catch.

Exemplul1 (Tratarea exceptiilor)

```
using System;
class ExceptieDemo
{
   public static void Main()
   {
      double[ ] vector = new double[4];
      int a = 1, i = -1;
      do
      {
         try
         {
            i++;
            vector[i] = a / i;
         }
         catch (DivideByZeroException exc)
         {  Console.WriteLine("S-a produs exceptia " + exc.Message);  }
         catch (IndexOutOfRangeException ex)
         {  Console.WriteLine("S-a produs exceptia " + ex.Message);    }
      }
      while (i < 4);
   }
}
```

Rezultat:

S-a produs exceptia Attempted to divide by zero.
S-a produs exceptia Index was outside the bounds of the array.

Programul de mai sus genereaza doua exceptii la rularea sa. O prima exceptie se produce atunci cand variabila intreaga i ia valoarea zero (DivideByZeroException), in timp ce cea de-a doua se produce atunci cand i=4 (IndexOutOfRangeException). Ambele exceptii sunt tratate de instructiunile catch corespunzatoare.

Interceptarea tuturor exceptiilor

Pentru a intercepta toate exceptiile, indiferent de tipul lor, utilizati un catch fara parametru sau, avand in verere ierarhia claselor care reprezinta exceptii, un catch avand ca parametru clasa Exception). Exemplul de mai jos, instructiunea catch intercepteaza atat exceptia DivideByZeroException cat si exceptia IndexOutOfRangeException.

Exemplul 2 (Interceptarea tuturor exceptiilor)

```
using System;
class ExceptieDemo
{
   public static void Main()
   {
      double[ ] vector = new double[4];
      int a = 1, i = -1;
      do
      {
         try
         {
            i++;
            vector[i] = a / i;
         }
         catch
         {  Console.WriteLine("S-a produs o exceptie " );  }
      }
      while (i < 4);
   }
}
```

Rezultat:

S-a produs o exceptie
S-a produs o exceptie

Imbricarea blocurilor try

Este posibil ca un bloc try sa fie continut in interiorul altuia. Exceptiile generate de blocul try interior si care nu sunt interceptate de instructiunile catch asociate acelui bloc sunt transmise blocului try exterior si deci pot fi interceptate de instructiunile catch asociate blocului try exterior. In exemplul de mai jos, exceptia IndexOutOfRangeException este lansata in blocul try interior, insa este interceptata de cel exterior.

Exemplul 3 (Imbricarea blocurilor try)

```
using System;
class ExceptieDemo
{
   public static void Main()
   {
      double[ ] vector = new double[4];
      int a = 1, i = -1;
      try       //exterior
      {    do
           {
              try  //interior
              { i++;  vector[i] = a / i; }
              catch (DivideByZeroException exc)
              { Console.WriteLine("S-a produs exceptia " + exc.Message); }
           }
           while (i < 4);
      }
      catch (IndexOutOfRangeException exc)
      {   Console.WriteLine("S-a produs exceptia " + exc.Message);
          Console.WriteLine("Eroare critica-programul se termina");  }
   }
}
```

Rezultat:

S-a produs exceptia Attempted to divide by zero.
S-a produs exceptia Index was outside the bounds of the array.
Eroare critica-programul se termina

Blocurile try imbricate se utilizeaza pentru a permite tratarea diferentiata a categoriilor diferite de erori. Unele tipuri de erori sunt critice si nu pot fi remediate, in timp ce altele sunt minore si pot fi tratate imediat in corpul instructiunilor catch. Un bloc try exterior intercepteaza o instructiune severa, in timp ce blocurile interioare erori minore care pot fi tratate.

Lansarea unei exceptii

Exemplele precedente intercepteaza exceptii generate automat de C#.

Este insa posibil sa lansam manual o exceptie, utilizand instructiunea throw a carei forma generala este:

throw exOb;

exOb trebuie sa fie un obiect al carui tip este o clasa derivata din Exception.

In programul alaturat este lansata o exceptie de tipul InsufficientMemoryException

Remarcati modul in care a fost produsa exceptia. Intrucat o exceptie este un obiect, trebuie utilizat operatorul new pentru a crea obiectul instanta al clasei InsufficientMemoryException.

Exemplul 4 (Lansarea manuala a unei exceptii)

```
using System;

class ThrowDemo
{
   public static void Main()
{
      try
   {
      Console.WriteLine("Inainte de throw");
      throw new InsufficientMemoryException();
   }
   catch (InsufficientMemoryException)
   {
      Console.WriteLine("Exceptie
    interceptata");
   }
}
}
```

Rezultat:

Inainte de throw
Exceptie interceptata

Relansarea unei exceptii

O exceptie interceptata de o instructiune catch poate fi relansata, cu scopul de a fi interceptata de un bloc catch exterior. Exceptiile sunt relansate pentru a fi tratate de mai multe rutine de tratare. De exemplu, o prima rutina poate trata un aspect al exceptiei, in timp ce celelalte aspecte sunt tratate de o rutina aflata intr-un alt bloc catch. Pentru a relansa o exceptie se utilizeaza instructiunea throw fara a specifica vreo exceptie, ca in exemplul de mai jos. Exceptia relansata de blocul interior va fi interceptata de blocul exterior.

Exemplul 5 (Relansarea unei exceptii)

```
using System;
class ExceptieDemo
{    public static void Main()
   {   double[ ] vector = new double[4];
       int a = 1, i = -1;
       try       //exterior
       {   do
          {   try  //interior
              { i++; vector[i] = a / i; }
              catch (DivideByZeroException exc)
              { Console.WriteLine("S-a produs exceptia " + exc.Message); }
              catch (IndexOutOfRangeException exc)
              {   Console.WriteLine("Exceptia: " + exc.Message+ " In blocul try interior ");
                  throw; }  //relansarea exceptiei
          }
          while (i < 4);
       }
       catch (IndexOutOfRangeException exc)
       {    Console.WriteLine("Exceptia: " + exc.Message+" Relansata");   }
   }
}
```

Rezultat:

S-a produs exceptia Attempted to divide by zero.
Exceptia: Index was outside the bounds of the array. In blocul try interior
Exceptia: Index was outside the bounds of the array. Relansata

Utilizarea lui finally

In unele situatii este necesar ca un bloc de instructiuni sa se execute la terminarea unui bloc try/catch. Spre exemplu, o exceptie care se produce in program cauzeaza terminarea prematura a unei metode care a deschis un fisier sau o conexiune la retea care trebuie inchisa. Limbajul C# pune la dispozitie o modalitate convenabila de tratare a acestor probleme prin intermediul blocului finally.

Pentru a specifica un bloc de cod care trebuie sa se execute neaparat la iesirea dintr-o secventa try/catch, fie ca blocul try s-a executat normal fie ca s-a terminat datorita unei exceptii, se utilizeaza un bloc finally. Forma generala a unui bloc try/catch care include un bloc finally este:

```
try { // bloc de cod monitorizat pentru detectarea erorilor la executie}
catch(TipExceptie1 exOb){ //rutina de tratare pentru TipExceptie1}
catch(TipExceptie2 exOb){ //rutina de tratare pentru TipExceptie2}
 ……..
catch(TipExceptieN exOb){ //rutina de tratare pentru TipExceptieN}
finally{ //bloc final}
```

Clasa Exception

Asa cum am vazut toate exceptiile sunt derivate din clasa Exception. Aceasta clasa defineste mai multi constructori, metode si cateva proprietati. Trei dintre cele mai interesante proprietati sunt Message, StackTrace si TargetSite. Toate trei sunt accesibile numai la citire. Message contine un string care descrie natura erorii, StackTrace contine un alt string care prezinta stiva apelurilor care a condus la aparitia exceptiei. TargetSite intoarce un obiect care specifica metoda care a generat exceptia. Dintre metodele clasei Exception, amintim metoda ToString() care intoarce un string care descrie exceptia. Metoda ToString() este apelata automat atunci cand exceptia este afisata prin intermediul lui WriteLine().

Exemplul 6 (Utilizarea proprietatilor clasei Exception)

```
using System;
class ExceptieDemo
{    public static void Main()
   {   double[ ] vector = new double[5];
       int a = 1, i = -1;
       do
       {   try
          {   i++;        vector[i] = a / i;       }
          catch(DivideByZeroException exc)
          {   Console.WriteLine("S-a produs o exceptie ");
              Console.WriteLine("Mesajul standard este: \n" + exc);
              Console.WriteLine("\nStiva de apeluri: " + exc.StackTrace);
              Console.WriteLine("Mesajul: " + exc.Message);
              Console.WriteLine("Metoda care a generat exceptia: " + exc.TargetSite);     }
       }
       while (i < 4);
   }
}
```

Rezultat:

```
S-a produs o exceptie
Mesajul standard este: System.DivideByZeroException: Attempted to divide by zero.
   at ExceptieDemo.Main() in H:\C#appl\Project14\Project14\CodeFile1.cs:line 16
Stiva de apeluri:    at ExceptieDemo.Main() in H:\C#appl\Project14\Project14\CodeFile1.cs:line 16
Mesajul: Attempted to divide by zero.
Metoda care a generat exceptia: Void Main()
```

Clase derivate din Exception

Erorile cel mai frecvent intalnite sunt tratate utilizand exceptiile predefinite in C#. Insa, mecanismul de tratare a exceptiilor nu se limiteaza doar la acestea. Utilizatorul poate defini exceptii personalizate pentru tratarea erorilor specifice care pot aparea in programele sale.

Pentru a crea un nou tip de exceptii este suficient sa definiti o clasa derivata din Exception. Ca regula generala (insa nu obligatorie), exceptiile definite de utilizator trebuie sa fie derivate din clasa ApplicationException, aceasta fiind radacina ierarhiei rezervate pentru exceptiile generate de aplicatii. Clasele derivate de utilizator nu trebuie sa implementeze nimic, simpla lor apartenenta la ierarhia de tipuri permite utilizarea lor ca exceptii.

Clasele create de utilizator dispun, prin mostenire, de proprietatile si metodele definite de clasa Exception. Unul sau mai multi membrii ai clasei Exception pot fi extinsi in clasele reprezentand exceptii pe care utilizatorul le poate crea.

Spre exemplu, programul urmator creaza o exceptie numita NonIntResultException. Aceasta exceptie este lansata in program atunci cand impartirea a doua valori intregi produce un rezultat cu o parte fractionara. Clasa NonIntResultException defineste doi constructori standard si redefineste metoda ToString().

Exemplu 7 (Clasa derivata din Exception)

```
using System;
class NonIntResultException: ApplicationException
{
    public NonIntResultException()
        : base()
    { }
    public NonIntResultException(string str)
        : base(str)
    { }
    public override string ToString()
   { return Message;}

}
class ExceptieDemo
{
    public static void Main()
    {   int[ ] numarator = {2, -3, -2, 3, 47, 0, 23, 34, 23, 345, 48, 38, 34};
        int[] numitor = { 4, 0, -6, 3, 47, 0, 23, 34, 23 };
        for (int i = 0; i < numarator.Length; i++)
        {   try
            {   int a;
                a = numarator[i] / numitor[i];
                if (numarator[i] % numitor[i] != 0)
                { throw new NonIntResultException("Rezultatul "+ numarator[i]+ "/"+ numitor[i]+ " nu este intreg"); }
                Console.WriteLine("{0}/{1} este {2}", numarator[i], numitor[i], a);
            }
            catch(DivideByZeroException)
            { Console.WriteLine("Nu putem imparti prin zero"); }
            catch (IndexOutOfRangeException)
            { Console.WriteLine("Elementul nu exista in tabel"); }
            catch(NonIntResultException exc)
            { Console.WriteLine(exc); }
        }
    }
}
```

Rezultat:

Rezultatul 2/4 nu este intreg
Nu putem imparti prin zero
Rezultatul -2/-6 nu este intreg
3/3 este 1
47/47 este 1
Nu putem imparti prin zero
23/23 este 1
34/34 este 1
23/23 este 1
Elementul nu exista in tabel
Elementul nu exista in tabel
Elementul nu exista in tabel
Elementul nu exista in tabel

Interceptarea exceptiilor derivate

Daca trebuie sa interceptati exceptii avand ca tipuri atat o clasa de baza cat si o clasa derivata atunci clasa derivata trebuie sa fie prima in secventa de instructiuni catch. In caz contrar, instructiunea catch asociata clasei de baza va intercepta si exceptiile clasei derivate, conducand la existenta codului inaccesibil (codul din instructiunea catch asociata clasei derivate nu se va executa niciodata). *In C#, prezenta instructiunii catch inaccesibile este considerata eroare*. Programul de mai jos va genera eroare la compilare.

```
using System;
class NonIntResultException: ApplicationException
{    public NonIntResultException()
       : base()    { }
    public NonIntResultException(string str)
       : base(str)   { }
    public override string ToString()
   { return Message;}
}
class ExceptieDemo
{    public static void Main()
    {   int[ ] numarator = {2, -3, -2, 3, 47, 0, 23, 34, 23, 345, 48, 38, 34};
       int[] numitor = { 4, 0, -6, 3, 47, 0, 23, 34, 23 };
       for (int i = 0; i < numarator.Length; i++)
       {   try
          {   int a;   a = numarator[i] / numitor[i];
             if (numarator[i] % numitor[i] != 0)
             { throw new NonIntResultException("Rezultatul "+ numarator[i]+ "/"+ numitor[i]+ " nu este intreg"); }
             Console.WriteLine("{0}/{1} este {2}", numarator[i], numitor[i], a);
          }
          catch(DivideByZeroException)
          { Console.WriteLine("Nu putem imparti prin zero"); }
          catch (Exception)                                    //ordinea instructiunilor catch conteaza
          { Console.WriteLine("Elementul nu exista in tabel"); }  //programul genereaza o eroare la compilare
          catch(NonIntResultException exc)
          { Console.WriteLine(exc); }
       }          }     }
```