



Modulul 5

Vectori. Excepții

În acest curs se va prezenta modul de folosire a vectorilor în C#. Se vor discuta conceptele de bază ale vectorilor, incluzând declararea, accesarea și rangul unui vector. În a doua secțiune, se vor acoperi noțiunile privind crearea și inițializarea vectorilor. A treia secțiune va descrie modul de folosire a proprietăților și a metodelor vectorilor, modul de trimitere a acestora ca parametri, modul de întoarcere a unui vector de către o metoda, precum și folosirea instrucțiunii **foreach** pe vectori.

A doua temă a modulului o reprezintă mecanismele de tratare a excepțiilor. Mai exact, se va discuta modul în care se aruncă excepțiile și modul în care acestea sunt prinse și tratate, pentru a asigura funcționarea predictibilă a unui program.

Overview

- ▶ Vedere de ansamblu asupra vectorilor
- ▶ Crearea vectorilor
- ▶ Moduri de folosire a vectorilor
- ▶ Folosirea excepțiilor
- ▶ Ridicarea excepțiilor



2

➤ Vectorii oferă o metodă importantă pentru gruparea datelor. Pentru a folosi la maxim puterea limbajului C#, este important să se înțeleagă modurile de utilizare și de creare eficientă a vectorilor.

➤ La finalul acestui modul, veți putea:

- Să creați, să inițializați și să folosiți vectori de rang diferit
- Să înțelegeți legătura dintre o variabilă de tip vector și o instanță de vector
- Să folosiți vectori ca parametrii pentru metode
- Să întoarceți vectori din metode
- Să tratați și să prindeți excepții într-o aplicație C#



Vedere de ansamblu asupra vectorilor

- ▶ Ce este un vector?
- ▶ Notăția vectorilor în C#
- ▶ Rangul vectorilor
- ▶ Accesarea elementelor vectorilor
- ▶ Verificarea limitelor vectorilor
- ▶ Caracteristicile vectorilor

3

Această secțiune acoperă conceptele generale ale vectorilor, introduce sintaxa folosită pentru declararea vectorilor în C# și descrie proprietățile de bază, cum ar fi rangul sau accesarea elementelor.

În următoare secțiune, veți învăța cum să definiți și să folosiți vectorii.

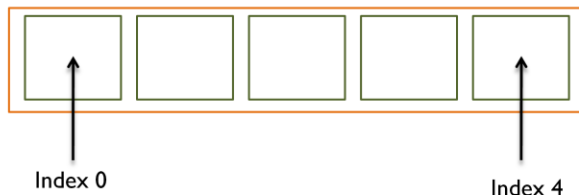


itacad
you@technology

Microsoft
.NET

Ce este un vector?

- ▶ Un vector este o înlanțuire de elemente
 - ▶ Metodă de a grupa date
 - ▶ Toate elementele din vector trebuie să fie de același tip
 - ▶ Elementele vectorului sunt accesate folosind indecși întregi



4

Există două metode fundamentale de a grupa date: structuri și vectori.

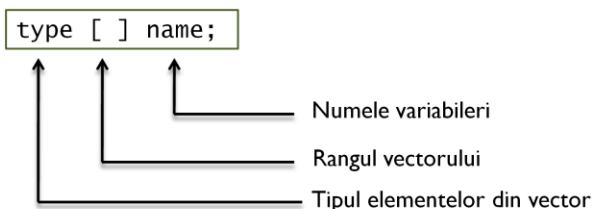
➤ **Structurile** sunt grupuri de date cu legătură între ele, dar de tipuri diferite. De exemplu, un nume (string), vârstă (int) și naționalitate (enum) se pot grupa într-o structură pentru a descrie o persoană. Membrii unei structuri pot fi accesați prin folosirea numelui câmpului.

➤ **Vectorii** sunt secvențe de date de același tip. De exemplu, o serie de case se grupează natural pentru a forma o stradă. Elementele dintr-un vector se accesează în funcție de un întreg de poziționare, numit index.

Notăția vectorilor în C#

► Variabilele de tip vector se declară specificând:

- Tipul de elemente din vector
- Rangul vectorului
- Numele variabilei



5

Vectorii se declară asemănător cu variabilele normale. În primul rând, trebuie precizat tipul elementelor ce vor alcătui vectorul, apoi o pereche de paranteze drepte [] pentru a marca faptul că se va declara un vector. Mai trebuie specificat doar numele variabilei, urmat de ‘;

Câteva restricții privind declararea vectorilor:

- Nu se pot pune parantezele drepte la dreapta numelui variabilei
- Nu se poate specifica dimensiunea vectorului la declarare.

Următoarele exemple arată ce este permis și ce nu în C#:

```
type[] name; //permis
type name[]; //NU este permis
type[4] name; //NU este permis
```

Rangul unui vector

- ▶ Cunoscut și ca dimensiune a vectorului
- ▶ Numărul de indecși asociați cu fiecare element

```
long [ ] vector;
```

Rang 1: Unidimensional
Un index este asociat cu
fiecare element

```
int [,] matrice;
```

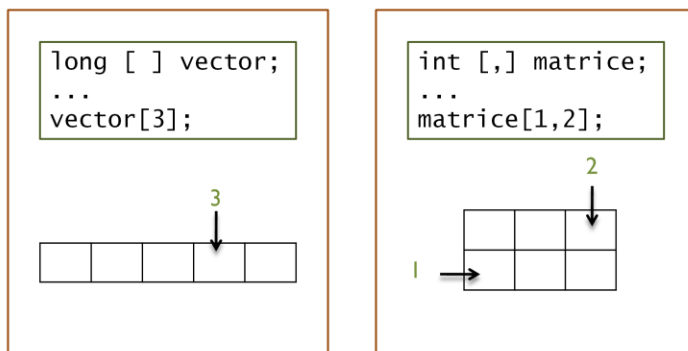
Rang 2: Bidimensional
Sunt necesari doi indecși
pentru a accesa un
element

Pentru a declara un vector unidimensional, se folosește o pereche de paranteze drepte pentru a marca rangul vectorului. Acesta mai este numit și vector de rang 1, fiind nevoie de un singur index pentru a identifica un element.

Pentru a declara un vector bidimensional, se pune o virgulă între parantezele drepte. Un astfel de vector este numit de rang 2, deoarece este nevoie de 2 indecși pentru a identifica un element. Acest mod de declarare se poate extinde, fiecare virgulă pusă între parantezele drepte va crește rangul vectorului cu unu.

Accesarea elementelor vectorilor

- ▶ Va trebui să oferiți câte un index pentru fiecare rang
- ▶ Toți indecșii încep de la 0



Pentru a accesa elemente ale vectorului, se folosește o sintaxă asemănătoare celei de la declararea vectorilor – în ambele cazuri se folosesc paranteze drepte. Această asemănare (care este voită și urmărește un curent popularizat de către C și C++) poate crea încurcături dacă nu sunteți familiari cu ea. De aceea, este important să faceți diferența între declararea unei variabile și expresia folosită pentru accesarea unui element.

Pentru a accesa un element într-un vector de rang 1, se folosește un index. Pentru a accesa un element într-un vector de rang 2, se folosesc doi indecși, separați printr-o virgulă. Notăția se extinde la fel ca pentru declararea vectorilor. Pentru a accesa un element dintr-un vector de rang k , se folosesc k indecși separați prin virgule.

Toți indecșii încep de la zero. Pentru a accesa primul element într-un vector unidimensional, folosiți expresia:

```
row[0]
```

și nu

```
row[1]
```

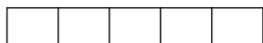
Indexarea de la 0 înseamnă că ultimul element al unui vector de dimensiunea $size$ se va afla la index $[size-1]$ și nu $[size]$.


 itacad
 you@technology

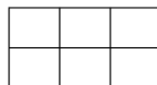
 Microsoft
.NET

Verificarea limitelor vectorilor

- ▶ Toate accesările elementelor unui vector se fac cu verificarea limitelor
 - ▶ Un index ce este în afara limitelor va arunca o excepție de tipul **IndexOutOfRangeException**
 - ▶ Folosiți proprietatea **Length** și metoda **GetLength** pentru a afla limitele unui vector



```
row.GetLength(0) == 5
row.Length == 5
```



```
grid.GetLength(0) == 2
grid.GetLength(1) == 3
grid.Length == 6
```

8

În C#, o expresie de acces la un element este automat verificată să asigure validitatea indexului. Această verificare a limitelor nu poate fi dezactivată.

Deși limitele sunt verificate, este recomandat să folosiți indecși corecți. Pentru a face asta, este recomandat să verificați manual limitele vectorilor, cel mai la îndemână fiind folosirea unui **for** și specificarea condiției de terminare a ciclării, ca în exemplul următor:

```
for (int i = 0; i < row.Length; i++)
{
    Console.WriteLine(row[i]);
}
```

Proprietatea **Length** întoarce numărul total de elemente din vector, indiferent de rangul vectorului. Pentru a vedea mărimea unei anumite dimensiuni, se folosește metoda **GetLength**, ca în exemplul următor:

```
for (int r = 0; r < grid.GetLength(0); r++)
{
    for (int c = 0; c < grid.GetLength(1); c++)
    {
        Console.WriteLine(grid[r,c]);
    }
}
```




Crearea Vectorilor

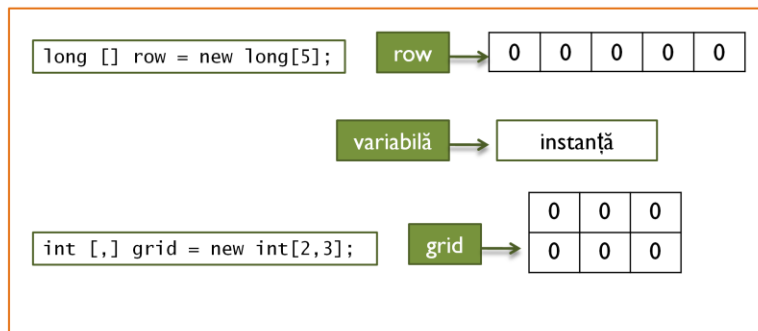
- ▶ Crearea de instanțe de vectori
- ▶ Inițializarea elementelor vectorilor
- ▶ Inițializarea vectorilor multi-dimensional
- ▶ Moduri de specificare a dimensiunii vectorilor
- ▶ Copierea variabilelor de tip vector

9

În această secțiune, veți învăța cum să creați instanțe de vectori, cum să inițializați aceste instanțe și cum să copiați variabile de tip vector.

Crearea de instanțe de vectori

- ▶ Declarația de variabile de tip vector **NU** creează un vector!
- ▶ Trebuie folosit cuvântul cheie **new** pentru a crea o instanță de vector
- ▶ Elementele vectorului vor fi inițializate la o valoare implicită



10

Declarația unui vector nu va crea efectiv acel vector. Acest lucru se datorează faptului că vectorii sunt tipuri referință și nu tipuri valoare. Trebuie folosit cuvântul cheie **new** pentru a crea instanța vectorului. În acest moment, trebuie specificată mărimea tuturor rangurilor, pentru crearea unei astfel de instanțe. Următoarele linii de cod vor genera erori de compilare:

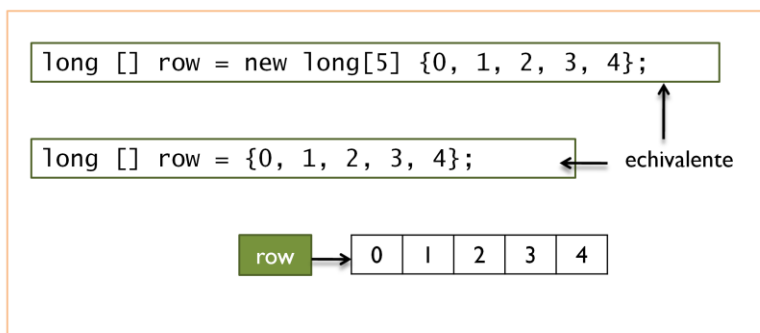
```
long[] row = new long[];           //nu este permis
int[,] grid = new int[,]           //nu este permis
```

Compilatorul C# va inițializa implicit toate elementele vectorului la o valoare standard, depinzând de tipul elementelor din vector. Pentru valori întregi, de exemplu, acestea vor fi inițializate la 0, pentru valori reale la 0.0, pentru valori booleane la **false**.

Memoria pentru vectori va fi întotdeauna alocată continuu, indiferent de tipul elementelor din vector și de dimensiunea sa. Din acest motiv, operațiunile de parcurgere de vectori vor fi foarte rapide.

Inițializarea elementelor vectorilor

- ▶ Elementele vectorilor pot fi inițializate explicit
 - ▶ Există și o variantă mai rapidă de a face declararea și inițializarea



11

Se poate folosi o expresie de inițializare pentru a seta unui vector alte valori inițiale decât cele implicite. O expresie de inițializare este compusă dintr-o serie de expresii, toate incluse între acolade și despărțite de virgule. Inițializarea este făcută de la dreapta la stânga, putând fi folosite pentru inițializare apeluri de metode și expresii complexe, ca în exemplul următor:

```
int[] data = new int[4] {a, b(), c*d, e() + f()};
```

În momentul în care inițializarea se face în aceeași linie de cod cu declararea, există posibilitatea de a folosi o variantă prescurtată a instanțierii și a inițializării:

```
int[] data1 = new int[4] {0, 1, 2, 3};    //permis
int[] data2 = {0, 1, 2, 3};              //permis
data2 = new int[4] {0, 1, 2, 3};         //permis
data2 = {0, 1, 2, 3};                    //NU este permis
```

La inițializarea vectorilor, trebuie specificate valorile pentru toate elementele vectorului. Nu se poate ca o parte a vectorului să fie inițializată și cealaltă parte să folosească valoarea implicită:

```
int[] data3 = new int[2] {};              //NU este permis
int[] data4 = new int[2] {42};           //NU este permis
int[] data5 = new int[2] {42, 42};       //permis
```

Inițializarea vectorilor multi-dimensionali

- ▶ Vectorii multi-dimensionali se pot inițializa asemănător
 - ▶ Toate elementele vectorului trebuie specificate

```
int [,] grid = {
    {5, 4, 3},
    {2, 1, 0}
};
```



```
int [,] grid = {
    {5, 4, 3}
    {2, 1 }
};
```



grid →

5	4	3
2	1	0

12

Discuția este asemănătoare ca la vectorii unidimensionali. Trebuie inițializate toate elementele vectorului.

Moduri de specificare a dimensiunii vectorilor

- ▶ Dimensiunea unui vector nu trebuie obligatoriu să fie o constantă disponibilă la compilare
 - ▶ Orice expresie ce întoarce un întreg va fi acceptată
 - ▶ Accesarea elementelor se va face la fel de rapid în toate cazurile

Specificarea dimensiunii la compilare

```
long [ ] row = new long[4];
```

Specificarea dimensiunii la run-time

```
string s = Console.ReadLine();
int size = int.Parse(s);
long [ ] row = new long[size];
```

13

Pentru a crea și instanța vectori multidimensionali, mărimea dimensiunilor poate fi calculată de către o expresie la run-time, nu trebuie să fie neapărat o constantă.

Se poate folosi chiar și o combinație între cele două:

```
System.Console.WriteLine("Numarul de RANDURI: ");
string s1 = System.Console.ReadLine( );
int rows = int.Parse(s1);
...
int[,] matrix = new int[rows,4];
```

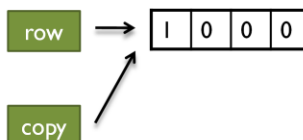
Există o singură restricție. Nu se poate folosi o expresie run-time pentru a specifica un vector ce este apoi inițializat:

```
string s = System.Console.ReadLine( );
int size = int.Parse(s);
int[ ] data = new int[size]{0,1,2,3};           //NU este permis
```

Copierea variabilelor de tip vector

- ▶ Se va face doar copierea variabilei
 - ▶ Nu se va face copierea instanței vectorului
 - ▶ Două variabile pot face referință la aceeași instanță de vector

```
long [ ] row = new long[4];
long [ ] copy = row;
...
row[0]++;
long value = copy[0]
Console.WriteLine(value);
```



14

La copierea unei variabile de tip vector, nu se va obține o nouă instanță a vectorului. Analizând codul de deasupra, se poate observa că ambele variabile fac referință la aceeași zonă de memorie.



Folosirea vectorilor

- ▶ Proprietăți și metode ale vectorilor
- ▶ Vectorii ca valori întoarse și ca parametri
- ▶ Argumente din linia de comandă
- ▶ Folosirea instrucțiunii **foreach** împreună cu vectori

15

În această secțiune, veți învăța cum să folosiți vectorii și cum să îi trimiteți ca parametri pentru metode.


 itacad
 you@technology

 Microsoft
 .NET

Proprietăți și metode ale vectorilor

► Proprietăți:

- **Rank** - proprietate read-only ce întoarce rangul vectorului
- **Length** – proprietate read-only ce întoarce numărul de elemente din vector

► Metode:

- **Sort** – va sorta un vector de rang 1
- **Clear** – va seta un interval de elemente la **0** sau **null**
- **GetLength** – va întoarce lungimea unei dimensiuni
- **IndexOf** – returnează indexul primei apariții a unui element căutat

16

Proprietăți

• **Rank** este o proprietate read-only care specifică rangul unei instanțe de vector. De exemplu, având următorul cod:

```
int[ ] one = new int[a];
int[, ] two = new int[a,b];
int[, , ] three = new int[a,b,c];
```

valorile pentru proprietatea **Rank** vor fi:

```
one.Rank == 1
two.Rank == 2
three.Rank == 3
```

• **Length** este o proprietate read-only ce specifică numărul de elemente din vector. De exemplu, pentru cele 3 definiții făcute mai devreme, valorile pentru **Length** vor fi:

```
one.Length == a
two.Length == a * b
three.Length == a * b * c
```



itacad
you@technology

Microsoft
.NET

Vectorii ca valori întoarse și ca parametri

- ▶ Variabilele de tip vector se trimit către metode și se returnează ca orice variabilă
- ▶ La trimiterea ca parametru, **NU** se va crea o instanță nouă

```
class Program {
    static void Main() {
        int[] row = CreateArray(20);
        GuessResult(row);
        Console.WriteLine(row[0]);
    }
    static int[] CreateArray(int size) {
        int[] created = new int[size];
        return created;
    }
    static void GuessResult(int[] row) {
        row[0]++;
    }
}
```

18

În acest slide, metoda CreateArray este implementată folosind două instrucțiuni. Cele două se pot combina într-o singură instrucțiune astfel:

```
static int[] CreateArray(int size)
{ return new int[size];}
```

Dimensiunea vectorului întors nu trebuie specificat. Dacă se încearcă acest lucru, se va obține un mesaj de eroare:

```
static int[4] CreateArray( ) {...} // Eroare la compilare
```

Asemănător, se pot întoarce și vectori de rang mai mare ca 1. În continuare, un astfel de exemplu:

```
static int[,] CreateArray( )
{
    string s1 = System.Console.ReadLine( );
    int rows = int.Parse(s1);
    string s2 = System.Console.ReadLine( );
    int cols = int.Parse(s2);
    return new int[rows,cols];
}
```

La trimiterea unui vector ca parametru al unei metode, vectorul devine accesibil metodei. Astfel, vectorul accesibil în metodă este, de fapt, un pointer către aceeași instanță a vectorului. Modificările făcute în metoda GuessResult asupra vectorului row vor fi văzute în metoda Main.

Datorită faptului că trimiterea unei variabile de tip vector ca parametru nu crează o nouă instanță, această trimitere este foarte rapidă.



Argumente din linia de comandă

- ▶ La run-time, se vor trimite argumentele în linie de comandă metodei **Main**
 - ▶ Argumentele se vor trimite sub forma unui vector de stringuri
 - ▶ Numele programului nu este membru al acestui vector

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i < args.Length; i++)
        {
            System.Console.WriteLine(args[i]);
        }
    }
}
```

19

La rularea unei aplicații pentru consolă, se pot oferi argumente suplimentare din linie de comandă. De exemplu, dacă rulați programul **pkzip** (utilitar de arhivare) din linie de comandă, puteți oferi argumente suplimentare pentru a controla crearea fișierelor .zip . Următoarea comandă adaugă recursiv toate fișierele *.cs într-o arhivă:

```
C:\> pkzip -add -rec -path=relative c:\code *.cs
```

Dacă ați fi scris programul **pkzip** folosind C#, ați fi prins aceste argumente din linia de comandă ca un vector de string-uri care este trimis la run-time metodei **Main**:

```
class PKZip
{
    static void Main(string[] args)
    {
        ...
    }
}
```

În acest exemplu, când se rulează programul **pkzip**, la runtime se execută efectiv următoarele instrucțiuni:

```
string[] args = {"-add", "-rec", "-path=relative",
"c:\\code", "*.cs"};
PKZip.Main(args);
```


itacad
you@technology

Microsoft
.NET

Folosirea instrucțiunii **foreach** împreună cu vectori

- ▶ Exemplul anterior se poate rescrie foarte ușor folosind instrucțiunea **foreach**

```
class Program
{
    static void Main(string[] args)
    {
        foreach(string argument in args)
        {
            Console.WriteLine(argument);
        }
    }
}
```

20

foreach este o instrucțiune foarte puternică, pentru că face instrucțiunile de iterație transparente pentru programator. Fără instrucțiunea **foreach**, codul ar arăta așa:

```
for (int i = 0; i < args.Length; i++)
{
    System.Console.WriteLine(args[i]);
}
```

Folosind **foreach**, codul devine:

```
foreach (string arg in args)
{
    System.Console.WriteLine(arg);
}
```

Observați că, în momentul în care se folosește instrucțiunea **foreach**, nu mai este nevoie de:

- un întreg folosit pentru index (int i)
- verificarea limitelor vectorului (i < args.Length)
- o expresie de acces la elementul vectorului (args[i])

Se poate folosi **foreach** și pentru a itera pe vectori cu rang mai mare de 1. De exemplu, următorul cod va avea ca rezultat afișarea: 0 1 2 3 4 5

```
int[,] numbers = { {0,1,2}, {3,4,5} };

foreach (int number in numbers)
{
    System.Console.WriteLine(number);
}
```

Caracteristicile vectorilor

- ▶ Un vector nu se poate redimensiona când este plin
- ▶ Un vector este folosit pentru a grupa un tip de obiecte
- ▶ Elementele unui vector nu pot fi accesate read-only
- ▶ În general, vectorii sunt rapizi, dar nu flexibili

Dimensiunea și tipul elementelor

Dimensiunea unui vector și tipul elementelor sale sunt fixate permanent la creare. Vectorul nu se va putea mări sau micșora și nu va putea conține niciodată alte tipuri de elemente.

Proprietatea ReadOnly

Folosind cuvântul cheie **readonly** se pot scoate drepturile de scriere asupra unei variabile. Totuși, aplicat unui vector, codul va genera eroare la compilare. Următoarele două exemple vor genera eroare la compilare

```
const int [] array = {0, 1, 2, 3};
readonly int[] array = {4, 2};
```

Identificați greșala

- ▶ `int[] array;
array = {0, 2, 4, 6};`
- ▶ `int[] array;
System.Console.WriteLine(array[0]);`
- ▶ `int[] array = new int[3];
System.Console.WriteLine(array[3]);`
- ▶ `int[] array = new int[];`
- ▶ `int[] array = new int[3] {0, 1, 2, 3};`

- 1) Varianta prescurtată de instanțiere și inițializare este posibilă doar dacă este făcută într-o singură instrucțiune.
- 2) Variabila a fost declarată, dar vectorul nu a fost instanțiat.
- 3) Indexul 3 referă al 4-lea element într-un vector de 3 elemente. Se încercă accesarea unui element ce nu aparține vectorului.
- 4) În momentul instanțierii unui vector, trebuie precizată mărimea acestuia.
- 5) Dimensiunea vectorului a fost setată la 3 (prin `new int [3]`), dar la inițializare au fost furnizate 4 elemente.



Folosirea excepțiilor

- ▶ De ce excepții?
- ▶ Obiectele de tip Exception
- ▶ Folosirea blocurilor try și catch
- ▶ Blocuri catch multiple



23

Ca programator, cu siguranță vi se pare, câteodată, că o mai mare parte din timp este petrecut verificând dacă există erori și corectându-le, decât programând efectiv. Puteți rezolva această problemă folosind excepții. Excepțiile au fost gândite pentru a trata erori. În continuare, vom discuta despre cum se prind și cum se aruncă excepțiile în C#.


itacad
you@technology

Microsoft
.NET

De ce excepții?

- ▶ Metodele de tratare a erorilor în programarea procedurală sunt greoaie și neintuitive
- ▶ Logica programului și tratarea erorilor se face în același loc

```
int errorCode = 0;
FileInfo source = new FileInfo("code.cs");
if (errorCode == -1) goto Failed;
int length = (int)source.Length();
if (errorCode == -2) goto Failed;
// Success :)
Failed: ...
```

24

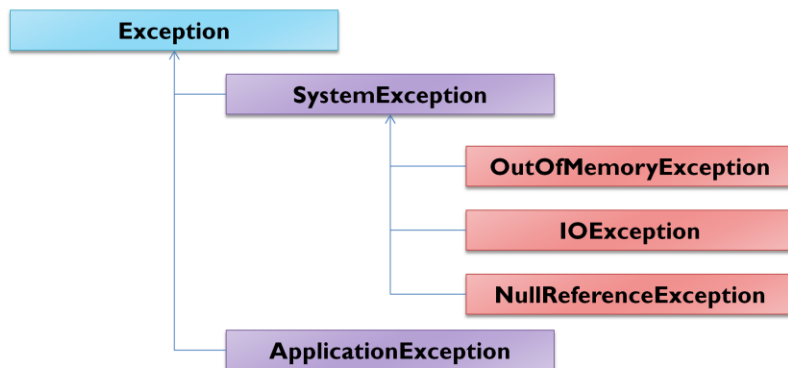
Un programator care are în vedere eventualele evenimente neașteptate ce pot să apară și care ia măsuri ca aplicația să trateze acest comportament și să funcționeze în continuare, va scrie cod robust, de bună calitate. Erori pot să apară la orice moment, fie la compilare, fie la execuția unui program.

Totuși, metodele tradiționale de tratare a erorilor au neajunsuri foarte mari, logica programului putându-se pierde în codul de tratare a erorilor.

Pentru a rezolva aceste probleme, în C# se folosesc excepțiile pentru a trata erorile.

Obiectele de tip Exception

- ▶ Toate obiectele de tip Exception sunt derivate din clasa cu același nume.
- ▶ Clasa Exception este parte a CLR



25

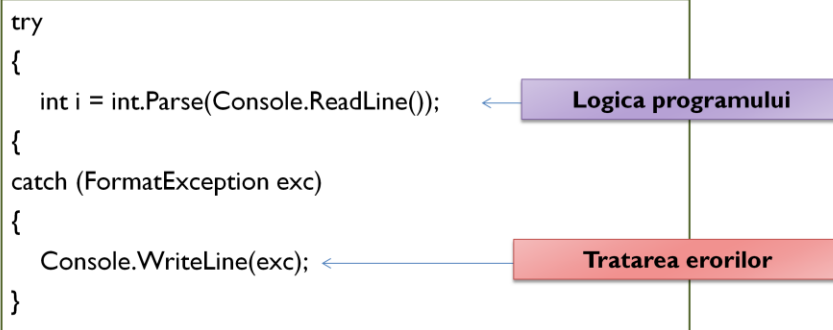
În metoda clasică de tratare a erorilor, au fost definite o serie de coduri de eroare pe care programatorul trebuie să le cunoască pentru a ști cauza apariției unei erori. Acest sistem însă s-a dovedit a fi complet neintuitiv, în plus, oferea detalii foarte puține despre eroarea ce a avut loc. De aceea platforma .NET a definit un set foarte mare de clase de excepții, foarte specifice.

Folosirea blocurilor try și catch

- ▶ Soluția POO pentru tratarea erorilor.
 - ▶ Codul se pune într-un bloc try
 - ▶ Eventualele excepții se tratează într-un bloc catch separat

```

try
{
    int i = int.Parse(Console.ReadLine());
}
catch (FormatException exc)
{
    Console.WriteLine(exc);
}
    
```



26

Programarea orientată pe obiecte oferă o soluție structurată pentru tratarea erorilor, sub forma blocurilor **try** și **catch**. Ideea este că se dorește separarea fizică între zona unde se scrie codul unei funcționări normale a programului și zona instrucțiunilor de tratare a erorilor. Așadar, bucățile de cod ce ar putea arunca excepții sunt plasate într-un bloc **try** și codul pentru tratarea eventualelor excepții, separat, într-un bloc **catch**.

Blocuri catch multiple

- Fiecare bloc catch se ocupă cu un anumit tip de excepție
- Un bloc try poate avea un singur bloc catch general asociat
- Nu se poate prinde o excepție ce este derivată dintr-o altă excepție prinsă anterior

```
try
{
    int i = int.Parse(Console.ReadLine());
    int j = int.Parse(Console.ReadLine());
    int k = i / j;
}
catch (FormatException exc) {...}
catch (DivideByZeroException exc) {...}
```

27

Blocul **try** poate conține oricâte instrucțiuni, fiecare dintre acestea putând ridica excepții de tipuri diferite. De aceea, pot exista mai multe blocuri **catch**, fiecare prinzând un tip specific de excepție.

O excepție este prinsă doar pe baza tipului său. La runtime, excepțiile sunt prinse automat în blocurile care tratează excepțiile respective.

În momentul în care se scrie cod în blocul **try**, nu există grija dacă o instrucțiune anterioară a eșuat. Dacă acest eveniment s-a petrecut, atunci controlul va fi dat unui bloc **catch** și nu va mai ajunge la instrucțiunea curentă.

Dacă un bloc potrivit pentru tratarea excepției nu se găsește, atunci este oprită execuția metodei curente și se încearcă găsirea unui bloc **catch** corespunzător în locul de unde a fost apelată metoda.

Bloc catch general

Un bloc **catch** general poate prinde orice excepție, indiferent de clasa sa. Este des folosit pentru a prinde excepții ce altfel nu ar fi prinse, datorită faptului că tratarea nu a fost corect făcută.

```
catch (Exception ex) {...}
```



itacad
you@technology

Microsoft
.NET

Ridicarea excepțiilor

- ▶ Instrucțiunea throw
- ▶ Clauza finally
- ▶ Verificarea depășirii limitei (overflow) în operații aritmetice
- ▶ Sfaturi pentru folosirea excepțiilor



28

C# pune la dispoziție instrucțiunea **throw** și clauza **finally** pentru ca programatorii să poată ridica excepții și să le trateze așa cum trebuie.

După terminarea acestei lecții, veți putea să:

- Aruncați propriile excepții
- Activați verificarea pentru operații aritmetice

Instrucțiunea throw

- ▶ Aruncă o anumită excepție
- ▶ Suspendă execuția normală a programului
- ▶ Controlul este dat primului bloc catch care prinde excepția care a fost aruncată

```
if (minute < 0 || minute >= 60)
{
    throw new InvalidTimeException(minute + " is not a valid time");
    // Nu se mai ajunge cu execuția aici
}
```

29

Blocurile **try** și **catch** sunt folosite pentru a prinde excepții aruncate de programe C#. Ați văzut cum, în loc să se întoarcă o valoare specifică în caz de eroare, C# cauzează execuția unui bloc **catch** în cazul unei excepții. Excepțiile conțin, de asemenea, și un mesaj de tip string în care pot fi trecute detalii importante privind starea programului în momentul în care a apărut excepția.

Aceste excepții pot fi însă aruncate și de către programator, dacă acesta vrea să anunțe o altă zonă a codului că o anumită operație nu s-a efectuat cu succes. Execuția normală a programului se va suspenda imediat și controlul va fi trimis primului bloc **catch** întâlnit care poate trata tipul de excepție aruncată.

Obs: în momentul aruncării excepției, va trebui declarat un nou obiect de tipul excepției dorite. De aceea, se va folosi cuvântul cheie **new** pentru a realiza acest lucru.

Tipurile ce pot fi aruncate

Nu se pot arunca decât obiecte ale căror tip este derivat din **System.Exception**. Acest lucru este diferit de C++, unde orice obiect se poate arunca.

Se poate folosi o instrucțiune **throw** într-un bloc **catch** pentru a rearunca excepția ce a fost prinsă:

Clauza finally

- ▶ Toate instrucțiunile dintr-un bloc **finally** sunt întotdeauna executate
- ▶ Chiar dacă programul iese dintr-un bloc **try** pentru a trata o eroare, se va întoarce pentru a executa instrucțiunile din blocul **finally**

```
try {...}
```

```
catch () {...}
```

```
finally {...}
```

Se va ajunge aici, indiferent dacă se întâlnesc excepții sau nu

30

C# oferă clauza **finally** pentru a marca instrucțiuni ce trebuie executate, indiferent dacă execuția a întâlnit sau nu excepții. Astfel, dacă controlul programului iese dintr-un bloc **try**, ca rezultat al unei execuții normale, instrucțiunile din blocul **finally** vor fi executate. De asemenea, dacă controlul părăsește blocul **try** ca rezultat al apariției unei excepții sau a unei instrucțiuni de salt (**break**, **continue**, **goto**), instrucțiunile din blocul **finally** vor fi de asemenea executate.

Această clauză este utilă în două situații: să nu se scrie instrucțiuni de două ori și pentru eliberarea anumitor resurse după ce o excepție a fost aruncată.

Obs: Se va raporta eroare dacă instrucțiunile de salt (**break**, **continue**, **goto**, **return**) încearcă să transfere controlul programului în afara unui bloc **finally**. **Break**, **continue**, **goto** au voie să fie folosite dacă, după salt, controlul programului se va afla în același bloc **finally**. Instrucțiunea **return** nu are însă voie să fie folosită, nici dacă este ultima instrucțiune dintr-un bloc **finally**


itacad
you@technology

Microsoft
.NET

Verificarea operațiilor aritmetice

- ▶ În mod implicit, astfel de verificări nu au loc
- ▶ Comportamenul se poate schimba global la compilare
- ▶ Instrucțiunea `checked` activează verificarea pentru un anumit bloc

```
checked
```

```
{
    int number = int.MaxValue;
    Console.WriteLine(number++);
}
```

OverflowException

Se aruncă excepție
WriteLine **NU** se execută

```
unchecked
```

```
{
    int number = int.MaxValue;
    Console.WriteLine(number++);
}
```

-2147483648

MaxValue + 1 e negativ???


31


În mod implicit, operațiile matematice nu sunt verificate pentru a vedea dacă limitele pentru valorile ce pot fi memorate sunt depășite. Totuși, sunt multe situații în care acest lucru este util.

Crearea unor expresii `checked` și `unchecked`

Cuvintele cheie `checked` și `unchecked` pot fi folosite și pentru a specifica verificarea unor singure expresii:

```
static void Main()
{
    int nr = int.MaxValue;
    Console.WriteLine(unchecked(++number)); //va afișa un număr negativ
    Console.WriteLine(checked(++number)); //va arunca excepție
}
```


itacad
 you@technology



Sfaturi pentru folosirea excepțiilor

- ▶ La aruncare:
 - ▶ Nu folosiți excepții pentru cazurile normale sau așteptate
 - ▶ Includeți un string de descriere în instanța excepției
 - ▶ Folosiți cele mai specifice clase posibile
- ▶ La prindere:
 - ▶ Aranjați blocurile catch de la cel mai specific la cel mai general
 - ▶ Nu lăsați excepțiile să iasă din funcția Main

32

Nu folosiți excepții pentru cazurile normale sau așteptate

Includeți un string de descriere în instanța excepției

Întotdeauna includeți o descriere folositoare într-un obiect excepție. Acest string poate fi folosit în momentul prinderii.

Folosiți cele mai specifice clase posibile

Se pot afla informații mai multe dintr-o excepție mai specifică decât dintr-una generală. Ca exemplu, aruncați **FileNotFoundException** decât mai generala **IOException**.

Aranjați blocurile catch de la cel mai specific la cel mai general

Dacă veți face prinderea invers, la blocurile **catch** specifice nu se va mai ajunge niciodată, acestea fiind prinse de către blocurile generale.

Nu lăsați excepțiile să iasă din funcția Main.

La ieșirea din Main, puneți o clauză **catch** generală, pentru a prinde toate excepțiile ce nu au fost tratate.



Overview

- ▶ Vedere de ansamblu asupra vectorilor
- ▶ Crearea vectorilor
- ▶ Moduri de folosire a vectorilor
- ▶ Folosirea excepțiilor
- ▶ Ridicarea excepțiilor



33