



Modulul 1

Tipuri și interfețe standard

Overview

- Tipuri valoare si referință
- Utilizarea tipurilor generice
- Implementarea interfețelor standard
- Implementarea delegaților și evenimentelor
- Excepții



Microsoft .NET Framework conține mii de tipuri de date predefinite care asigură o vastă funcționalitate aplicațiilor dezvoltate. **CTS (Common Type System)** cataloghează tipurile de date în tipuri de date valoare și tipuri de date referință. Acest modul descrie diferențele dintre aceste două categorii. De asemenea, este descris modul în care sunt create tipurile generice, modul în care sunt implementate interfețele, utilizarea delegaților și a evenimentelor, și utilizarea excepțiilor.

După completarea acestui modul studentul va dobândi următoarele cunoștințe:

- Utilizarea tipurilor predefinite valoare sau referință
- Implementarea tipurilor generice
- Implementarea interfețelor, în felul acesta impunând componentelor să respecte anumite standarde
- Controlul interacțiunii dintre componentele unei aplicații utilizând delegați și evenimente
- Definirea și utilizarea excepțiilor

Definirea și utilizarea tipurilor valoare

- ▶ Tipuri valoare în .NET Framework:
 - ▶ **int, float, double, bool, char**
 - ▶ **structuri și enumerații**
- ▶ Tipurile valoare sunt copiate prin valoarea acestora
 - ▶ Metodele primesc o copie a acestora
- ▶ Reguli de alocare și dealocare
 - ▶ Alocare pe stivă
 - ▶ Dealocarea atunci când variabila iese din domeniul de vizibilitate (ex.: bloc **for**, metodă)

Academia Microsoft

Majoritatea limbajelor de programare dispun de tipuri de date predefinite precum nr. întregi sau reale (int și float) care sunt copiate prin valoare atunci când sunt transmise într-o metodă. În *.NET Framework* acestea sunt numite tipuri valoare. CLR suportă două feluri de tipuri valoare:

- Predefinite: *System.Int32*(int), *System.Double*(double)
- Definite de utilizator: structuri și enumerații

Când este creată o instanță a unui tip valoare, memoria este alocată pe stivă, și este automat eliberată la sfârșitul domeniului de utilizare (ex.: la sfârșitul metodei sau unui bloc logic). Invers, dacă o variabilă de tip valoare este declarată într-o clasă, memoria este alocată ca parte a blocului de memorie utilizat pentru obiect și este dealocată de către *garbage collector*.

Definirea structurilor

O structură este un tip de date valoare definit de către utilizator și care moștenește direct **System.ValueType**. De obicei se folosesc structuri atunci când se dorește reprezentarea unor tipuri de date numerice care sunt alocate și dealocate frecvent precum: coordonate, dimensiuni.

Pentru a crea o structură se folosește cuvântul cheie **struct**. Un tip structură poate conține variabile, metode, proprietăți și constructori parametrizați. Structurile pot implementa un număr oarecare de interfețe însă nu pot moșteni o clasă sau să fie părinte în procesul moștenirii.

Codul următor arată cum se definește o structură de reprezentare a coordonatelor. Structura din exemplu definește variabile, proprietăți și metode. Structura conține și un constructor parametrizat și supraîncarcă metoda **ToString**, moștenită din **System.ValueType**.

```
public struct Coordinate
{
    private int _x, _y;
    public Coordinate(int x, int y) ...

    public int X { get {...} set {...} }
    public int Y { get {...} set {...} }
    public void MoveTo(int x, int y) ...
    public override string ToString() ...
}
```

Definirea tipului enumerație

O enumerație este un tip de date valoare, definit de către utilizator care moștenește direct tipul **System.Enum**. De obicei se folosește enumerația pentru a reprezenta o mulțime fixată de opțiuni ca de exemplu zilele unei săptămâni. Pentru a defini o enumerație, se folosește cuvântul cheie **enum**. O enumerație conține mnemonici care sunt numere întregi ce încep de la 0 și cresc cu câte o unitate. Enumerațiile nu pot conține metode, proprietăți sau constructori, nu pot implementa interfețe și nu pot moșteni clase.

```
public enum Direction
{
    North,
    West,
    South,
    East
}
```

Definirea și utilizarea tipurilor referință

- ▶ Tipuri referință în .NET Framework:
 - ▶ **clase** și **delegați**
- ▶ Tipurile referință sunt copiate prin referința acestora
 - ▶ Metodele primesc o referință către obiectul original
- ▶ Reguli de alocare și dealocare
 - ▶ Alocare pe heap-ul programului
 - ▶ Dealocarea realizată de **garbage collector**

Academia Microsoft

Majoritatea tipurilor de date pe care le vom folosi în aplicațiile *.NET Framework* sunt tipuri referință: clase, interfețe și delegați. Când se crează o instanță a unui tip referință, memoria pentru acel obiect este alocată pe heap. Obiectul va rămâne alocat până când nu va mai exista nicio referință către acesta. Când GC va rula, va verifica mai întâi dacă nu mai există nicio referință către obiect și apoi va returna memoria heap-ului pentru a fi folosită de către alte obiecte.

Definirea claselor

Pentru a defini o clasă, se folosește cuvântul cheie **class**. Un tip clasă poate conține variabile, metode, proprietăți și constructori. O clasă poate moșteni o singură clasă și poate implementa orice număr de interfețe. Următorul exemplu de cod crează o clasă pentru a reprezenta un angajat. Observați că, aceasta este abstractă, prin urmare nu poate fi instanțiată, în schimb o puteți folosi prin derivare. Clasele derivate trebuie neapărat să implementeze metoda **Promote()** și pot reimplementa opțional metoda **CalculateBonus()**. (Polimorfism)

```
public abstract class Employee
{
    // State.
    private string _name;
    private DateTime _dateJoined;
    private decimal _salary;

    // Constructors.
    public Employee() ...
    public Employee(string n, DateTime d, decimal s) ...
    virtual decimal CalculateBonus() ➡ 5
}
```

```
// Properties.  
public string Name ...  
public DateTime DateJoined ...  
public decimal Salary ...  
// Methods.  
public abstract void Promote();  
public virtual decimal CalculateBonus() ...  
}
```

Definirea Interfețelor

Interfețele specifică un set de metode, proprietăți și evenimente pe care o clasă le poate implementa. Nu se poate crea o instanță a unei interfețe, în schimb se poate defini o clasă sau o structură care implementează această interfață.

```
public interface ILoggable  
{  
    bool DisplayTimestamps { get; set; }  
    void LogMessage(string message);  
    string[] GetMessages();  
}
```

Definirea delegaților

Un tip delegat încapsulează semnatura unei metode. Se poate crea o instanță a unui delegat care să refere o metodă cu o semnătură specifică. Ulterior, se poate invoca metoda la momentul dorit, utilizând instanța delegatului.


itacad
you@technology

Cum funcționează Garbage Collector?

- ▶ GC eliberează memoria utilizată de variabilele cu tipuri referință
 - ▶ Un obiect e distrus o singură dată, când nu mai e utilizat
- ▶ **Destructor** / metoda **Finalize**
 - ▶ Se vor apela înainte ca GB să distrugă obiectul
 - ▶ Nu vor fi apelate într-o anumită ordine
 - ▶ Se implementează interfața **IDisposable**



Academia Microsoft

Calculatoarele nu au cantități infinite de memorie, prin urmare memoria trebuie recuperată atunci când o variabilă sau un obiect nu o mai folosește. Obiectele de tipuri valoare sunt distruse imediat cum au ieșit din sfera de utilizare a programului. Acest lucru este realizat de către codul pe care compilatorul îl generează în momentul în care construim o aplicație și totul se întâmplă deterministic. În contrast, GB (parte a **CLR-ului**) reclamă memoria pe care variabilele referință o folosesc. Acest comportament este nedeterminist și nu este controlat direct de codul programatorului.

Procesul creării și distrugerii unui obiect

Crearea unui obiect se realizează utilizând cuvântul cheie **new**.

```
TextBox message = new TextBox(); // TextBox is a reference type
```

Din perspectiva programatorului, utilizarea lui **new** poate apărea ca o operație atomică, însă în spatele codului se execută de fapt două faze:

- Se alocă memorie brută pe heap. Programatorul nu are niciun control asupra acestei faze
- Se convertește blocul de memorie brută obținut la pasul anterior într-un obiect și are loc inițializarea obiectului. Aceasta fază care poate fi controlată de programator

După crearea unui obiect, se pot referi membrii săi (metode și variabile) și este posibil ca alte variabile referință să refere către acest obiect.

Se pot crea oricâte referințe către un anumit obiect. Aceasta influențează timpul de viață al obiectului. CLR trebuie să urmărească toate aceste referințe pentru că nu poate distruge obiectul original înainte ca toate referințele să fie șterse.

La fel ca și procesul de creare, procesul de distrugere este format din doi pași:

- Opțional, CLR execută niște operații pentru eliberarea resurselor pe care obiectul le-a utilizat. Se poate controla această fază utilizând un destructor.
- CLR dezalocă memoria pe care obiectul a folosit-o și o returnează heap-ului. Nu se poate controla aceasta etapă.

CLR execută cei doi pași utilizând **Garbage Collector**.

Scrierea destructorilor

Pentru a crea un destructor se folosește numele clasei prefixat de simbolul ~. Următorul exemplu arată o clasă care numără instanțele acesteia într-o variabilă statică:

```
class Tally
{
    public Tally()
    {
        ▶instanceCount++;
    }
    ~Tally()
    {
        ▶instanceCount--;
    }
    public static int InstanceCount()
    {
        ▶return instanceCount;
    }
    ...
    private static int instanceCount = 0;
}
```

Există câteva restricții foarte importante care se aplică destructorilor:

- Destructorii sunt valabili doar pentru tipurile referință, prin urmare nu puteți crea destructori într-o structură sau enumerație.
- Destructorii nu pot avea modificatori de acces și nu pot fi apelați explicit în cod (numai GB îi poate apela).
- Destructorul nu primește niciun parametru.

Compilatorul C# transformă automat destructorul într-o metodă suprascrisă a metodei **Object.Finalize**. Pentru exemplu de mai sus compilatorul transformă codul în ceva asemănător exemplului următor:

```
protected override void Finalize()
{
    try { ... }
    finally { base.Finalize(); }
}
```


Metoda **Finalize** pe care compilatorul C# o generează conține destructorul în interiorul corpului blocului **try**, urmat de un bloc **finally** ce apelează metoda **Finalize** a clasei de bază. Aceasta asigură că un destructor întotdeauna apelează destructorul clasei de bază. Este important să realizați că numai compilatorul poate face această translație. Nu puteți supraîncărca metoda **Finalize** și nu o puteți apela explicit în cod.

De ce folosim Garbage Collector?

Nu se poate distruge un obiect singur în C#. Motivele pentru care nu a fost lăsată această responsabilitate asupra programatorului sunt:

- Era uitată această etapă, ceea ce înseamnă ca memoria pe heap rămânea alocată și era foarte ușor să rămâneți fără memorie.
- Putea fi distrus un obiect activ. Nu uitați ca obiectele sunt accesate prin referința lor. Dacă un obiect menține o referință către un obiect care a fost distrus, va ajunge de fapt să refere fie o zonă de memorie nealocată, fie alt obiect.
- Se încercă distrugerea unui obiect mai mult decât o singură dată.

Toate aceste probleme nu mai apar când vine vorba de Garbage Collector.

Un lucru foarte important despre GB este că nu știm cu exactitate când acționează. Din cauză că acest proces este foarte costisitor, GB acționează doar atunci când rămânem fără memorie pe heap și colectează cât de multe obiecte poate.

Se poate invoca GB în program apelând metoda **System.GC.Collect**. Acest proces va rula asincron ceea ce înseamnă că chiar dacă programul nostru s-a terminat nu știm cu exactitate dacă obiectele au fost distruse. Utilizarea acestei metode nu este recomandată.

Cum funcționează GB?

GB rulează în propriul thread și numai atunci când este absolut nevoie. Cât timp rulează, celelalte thread-uri din aplicație vor intra în starea de halt. Aceasta se întâmplă pentru că s-ar putea ca GB să mute unele obiecte, prin urmare să schimbe referințe.

- Se construiește o hartă cu toate obiectele care sunt în aplicație în mod activ (fie direct, fie prin referințe către ele). Toate obiectele care nu se află în această hartă sunt distruse.
- Pentru fiecare obiect care urmează să fie distrus se verifică dacă există vreun destructor scris, caz în care este apelat. Dacă da este pus într-o coadă de finalizare.
- Se dezalocă memoria pentru aceste obiecte distruse mutând obiectele din hartă jos în heap. Această operațiune defragmentează heap-ul și eliberează memoria în vârful acestuia.
- În acest punct se vor relua thread-urile puse pe halt.
- Vor fi finalizate obiectele din coada de finalizare utilizând threadul propriu

Boxing / Unboxing

► Boxing

- O variabilă de tip valoare este asignată uneia de tip referință
- *Runtime* va crea o copie a obiectului valoare pe *heap*

► Unboxing

- Operația inversă "împachetării"
- *Runtime* crează o copie a obiectului pe stiva

.NET Framework ne permite să convertim un obiect de tip valoare într-un tip referință și apoi să îl reconvertim la tipul inițial. Conversia unei variabile de tip valoare în tip referință se numește **boxing**, iar operația inversă **unboxing**. Un dezavantaj al acestui mecanism este pierderea performanței.

Boxing

În momentul în care realizăm boxing, CLR crează o copie a variabilei pe heap și variabila de tip referință obținută va referi către această copie.

Se folosește în următoarele situații:

- Când trebuie să transmiteți unei metode un parametru care cere un *System.Object* sau un tip interfață.
- Când adaugați un obiect de tip valoare unei colecții nongenerice precum *System.Collections.ArrayList*

În următorul exemplu se crează o variabilă de tip *DateTime* (tip valoare) care este transmisă ca variabilă referință în momentul apelării metodei *Console.WriteLine*:

```
DateTime dt = DateTime.Now;
Console.WriteLine("Current date and time: {0}", dt);
```

Unboxing

Când se realizează conversia inversă, CLR verifică mai întâi dacă este validă, și apoi crează și returnează o copie a obiectului pe stivă.

Se folosește în următoarele scenarii:

- Când se folosește o metodă care returnează un *System.Object* sau un tip interfață și trebuie castat rezultatul la un tip valoare
- Când se obține un rezultat dintr-o colecție non-generică precum *System.Collections.ArrayList* care păstrează doar referințe *System.Object*.

```
// Create a collection.
ArrayList dates = new ArrayList();


// Create an object and add it to the collection. The
runtime boxes the object.
DateTime dt = new DateTime(2008, 1, 1);
dates.Add(dt);

// Retrieve the object from the collection and convert it
to a DateTime.
// The runtime unboxes the object to obtain a copy from
the heap.
DateTime copy1 = (DateTime)dates[0];

// Modify copy1 and display it. Displays January 2, 2008.
copy1 = copy1.AddDays(1);
Console.WriteLine("Copy1 date: {0}",
copy1.ToShortDateString());

// Retrieve the object from the collection again.
// The runtime unboxes the object to obtain another copy
from the heap
DateTime copy2 = (DateTime)dates[0];

// Modify copy2 and display it. Displays February 1, 2008.
copy2 = copy2.AddMonths(1);
Console.WriteLine("Copy2 date: {0}",
copy2.ToShortDateString());
```


itacad
 you@technology

Tipuri generice

- ▶ Este un tip exprimat sub forma unui parametru tip
 - ▶ Un parametru tip este un înlocuitor al tipului actual
 - ▶ Codul client va înlocui parametrul tip cu un tip explicit oferit ca argument
- ▶ Avantaje
 - ▶ Type Safety
 - ▶ Performanță
 - ▶ Algoritmi generici
- ▶ Utilizarea tipurilor generice

```
List<int> listOfNumbers = new List<int>();
```

Academia Microsoft

Tipurile generice au fost introduse în versiunea 2.0, și fac posibilă crearea de clase, structuri, interfețe și metode care amână specificarea unui tip sau chiar a mai multor tipuri până când codul client declară sau instanțiază clasa sau utilizează metoda. În acest fel se evita folosirea operațiilor de boxing și unboxing care sunt costisitoare.

Tipurile generice folosesc sintaxa **ClassName<T>**, unde **T** reprezintă tipul de date pe care clientul trebuie să îl specifice când folosește această clasă.

Avantajele folosirii tipurilor generice:

- **Type Safety** - tipurile generice obligă compilatorul să verifice tipurile
- **Performanță** - nu mai sunt necesare boxing și unboxing deoarece fiecare tip generic este specializat doar pentru un singur tip de date
- **Reutilizarea codului** – codul nu mai depinde de un singur tip de date în particular, prin urmare poate fi folosit în diverse situații
- **Algoritmi generici** – algoritmi abstracti care nu depind de un anumit tip sunt candidații perfecți pentru această categorie (de exemplu o procedură de sortare ce folosește interfața *IComparable*)

Exemplu de utilizare a tipurilor generice

Colecțiile generice sunt localizate în spațiul de nume **System.Collections.Generic**. Când se dorește crearea unei instanțe a unei clase generice, trebuie specificat tipul parametrilor pe care vrem să îi folosim. În exemplul următor avem o lista generică ce păstrează întregi și încă una ce păstrează șiruri de caractere:

```
List<int> listOfNumbers = new List<int>();
List<string> listOfWords = new List<string>();
```

Colecțiile generice mențin obiectele într-un mod mai eficient decât cele non-generice

Colecțiile non-generice păstrează referințe de tip *System.Object* către obiectele membrii, ceea ce înseamnă că CLR trebuie să realizeze *boxing* și *unboxing* de fiecare dată când un element e adăugat, respectiv șters.

Colecțiile generice mențin obiecte de un anumit tip. Prin urmare dacă acea colecție va păstra numere întregi referințele nu vor mai fi de tip *System.Object*, deci nu va avea loc *boxing/unboxing*.

Tipuri *nullable*

- ▶ Este un tip valoare care poate înlocui o valoare nulă
 - ▶ **Nullable<T>** sau **T?**
- ▶ Utilizarea tipurilor *nullable*

```
DateTime? dt;
...
if (dt.HasValue)
{
    DateTime v;
    v = dt.Value;
    ...
}
```

Academia Microsoft

Înainte de versiunea .NET 2.0 nu exista niciun mecanism prin care se putea specifica dacă un tip de date valoare are o valoare nedefinită cum exista **null** pentru tipurile referință.

Un tip valoare *nullable* păstrează o valoare definită sau valoarea **null**.

Utilizarea tipurilor *nullable*

Pentru a defini un tip valoare care are o valoare nulă predefinită se folosește construcția **Nullable<T>**:

```
Nullable<DateTime> myNullableDateTime;
```

// Visual C# also supports the following syntax for nullable variables.

```
DateTime? myNullableDateTime;
```

După declararea unei variabile *nullable*, îi puteți atribui o valoare de tipul respectiv (ex. *DateTime*) sau valoarea **null**:

```
myNullableDateTime = DateTime.Now; // Value assigned.
myNullableDateTime = null; // No value assigned.
```

Pentru a obține valoarea dintr-o variabilă *nullable*, folosiți proprietatea **HasValue** pentru a determina dacă variabila are o valoare definită și apoi proprietatea **Value**:

```
if (myNullableDateTime.HasValue)
{
    Console.WriteLine("{0}", myNullableDateTime.Value);
}
```

Definirea și utilizarea tipurilor generice

► Exemplu:

```
public struct Coordinate<T>
{
    private T _x, _y;
    public Coordinate(T x, T y) ...
    public void MoveTo(T x, T y) ...
    ...
}
```

```
Coordinate<int> c = new Coordinate<int>(5, 0);
c.MoveTo(100, 200);
```

Academia Microsoft

Definirea tipurilor generice

Pentru a specifica parametrul de tip pe care un tip generic o cere se folosește notația **<T>** la sfârșitul primei linii din definirea tipului, unde **T** este un înlocuitor pentru un tip de date. Se poate folosi orice nume pentru denumirea parametrului de tip (ex. T1, T2)

```
public class MyGenericClass<T>
{
    // Define class members here. Use T as a placeholder type
    // where necessary.
}
```

Se pot defini oricâți parametri de tip pentru un tip generic. Ex.: **<T1, T2, T3>**

Următorul exemplu de cod este un tip generic folosit pentru reprezentarea coordonatelor unui punct. Clientul poate folosi coordonate cu numere întregi, long, float sau alte tipuri care au sens în aplicație.

```
public struct Coordinate<T>
{
    // State.
    private T _x, _y;
    // Constructor.
    public Coordinate(T x, T y) ...
    // Properties.
    public T X ...
    public T Y ...
    // Methods.
    public void MoveTo(T x, T y) ...
    public override string ToString() ...
}
```

Utilizarea tipurilor generice

Când se folosește un tip generic în codul client trebuie ca fiecare parametru de tip să fie înlocuit cu un tip existent și explicit. Se pot declara variabile de tipul generic și le pot fi atribuite valori. Exemplu:

```
Coordinate<int> myIntCoord = new Coordinate<int>(5, 0);  
myIntCoord.MoveTo(100, 200);  
Console.WriteLine("{0}", myIntCoord.ToString());
```


Definirea metodelor generice

```
public class MyAlgorithms
{
    public static void DisplayArray<T>(T[] array)
    {
        foreach (T item in array)
            Console.WriteLine("{0}", item);
    }
}
```

```
string[] strArray = ...;
MyAlgorithms.DisplayArray(strArray);
```

Academia Microsoft

O metodă generică este o metodă care are cel puțin un parametru generic. Codul client va specifica tipul parametrului de fiecare dată când este apelată metoda. O metodă generică poate folosi un parametru de tip în lista de parametrii, ca tip de returnare sau în interiorul corpului acesteia.

Definirea unei metode generice

Pentru a specifica parametrul de tip al unei metode generice se adaugă **<T>** după numele metodei. Se pot defini și parametrii multipli de tip.

```
public class MyAlgorithms
{
    public static void DisplayArray<T>(T[] array)
    {
        foreach (T item in array)
        { Console.WriteLine("{0}", item) }
    }
}
```

Invocarea unei metode generice

Pentru a invoca o metodă generică în codul client, se folosește numele metodei și parametrii acesteia care sunt de un anumit tip. Dacă parametrii metodei nu specifică suficient parametrii de tip, atunci aceștia trebuiesc specificați explicit după numele metodei. Exemplu:

```
// Create an array of strings.
string[] myFriends = {"Claire", "Hayley", "Sara"};
// Invoke generic method, and specify type parameter implicitly.
MyAlgorithms.DisplayArray(myFriends);
// Invoke generic method, and specify type parameter explicitly.
MyAlgorithms.DisplayArray<string>(myFriends);
```

Constrângeri asupra tipurilor argumente

- ▶ Se pot defini constrângeri asupra tipurilor argumente pentru a specifica că acestea:
 - ▶ Trebuie să fie valoare
 - ▶ Trebuie să fie referință
 - ▶ Trebuie să aibă un constructor fără parametrii
 - ▶ Trebuie să fie o clasă specifică sau o subclasă a acestei clase
 - ▶ Trebuie să implementeze o interfață specifică
 - ▶ Trebuie să fie la fel sau să moștenească un al tip argument

Academia Microsoft

Când se definesc tipuri generice, de obicei se dorește ca un tip să fie cât mai independent posibil. Totuși, uneori este necesar să se asigure că argumentele de tip oferite de către codul client se încadrează în anumite constrângeri. De exemplu, dacă se dorește comparația a două obiecte pentru sortare, atunci acel tip trebuie să implementeze interfața *IComparable*. Dacă constrângerile impuse nu sunt respectate de codul client, va fi generată o eroare de compilare.

Definirea constrângerilor

| •Constrângere | •Descriere |
|-------------------------|---|
| •where T: struct | •Argumentul de tip trebuie să fie tip valoare |
| where T: class | •Argumentul trebuie să fie referință (clasă, interfață, delegat, array) |
| where T: new() | •Tipul trebuie să aibă un constructor accesibil și fără parametrii. Când e specificată împreună cu alte constrângeri trebuie să fie ultima. |
| where T: class_name | •Tipul trebuie să fie clasa specificată sau o subclasă a acestei clasei |
| where T: interface_name | •Tipul oferit ca argument trebuie să fie interfața specificată sau alt tip care implementează interfața specificată |
| where T: U | •Tipul trebuie să fie tipul oferit ca argument pentru U sau un tip derivat din acesta |

Se pot impune mai multe constrângeri pentru un parametru de tip dacă este nevoie. De exemplu se poate impune ca un anumit tip să implementeze mai multe interfețe și că trebuie să aibă un constructor fără parametrii.

Exemplu de definire a constrângerilor

Următorul exemplu arată cum trebuie definite constrângerile pentru un anumit parametru de tip. Sunt impuse următoarele condiții:

- Să fie de tip *Employee* sau subclasă a acestuia
- Implementează interfața *IDisposable* și interfața generică *IComparable*
- Are un constructor fără parametrii

```
public class EmployeeList<T>  
    where T : Employee, IDisposable, IComparable<T>, new()  
{  
    // Members.  
}
```


 itacad
 you@technology

Interfața **Comparable**

- ▶ Permite implementarea unui mecanism de comparare a două instanțe aparținând unui tip
 - ▶ Versiune generică și non-generică

```
public class Currency : Comparable<Currency>
{
    public int CompareTo(Currency other) ...
}
```

 Microsoft
.NET

Academia Microsoft

Interfața **Comparable** permite codului client să compare două instanțe ale unei clase sau ale unei structuri. Implementați această interfață, dacă doriți ca instanțele clasei dumneavoastră sau ale structurii să fie sortate. De exemplu, dacă doriți să sortați elementele dintr-o matrice prin metoda *Array.Sort*, elementele trebuie să fie de un tip care implementează interfața *Comparable*.

Versiunea 2.0 .NET a introdus o versiune generică a interfeței *Comparable* care lucrează cu obiecte cu tipuri generice. În contrast, interfața nongenerică *Comparable* lucrează cu tipul *System.Object*, ceea ce înseamnă că codul client poate transmite către comparație obiecte de diferite tipuri de date. Dacă implementați varianta nongenerică, trebuie să testați că obiectele primite sunt de același tip și aruncați *ArgumentException* în cazul în care acestea au tipuri diferite.

Interfața *Comparable* definește o singură metodă **CompareTo**, care compară instanța curentă cu un obiect de același tip transmis ca parametru. Aceasta returnează o valoare negativă dacă primul obiect este mai mic decât al doilea, 0 dacă sunt egale sau o valoare pozitivă dacă primul e mai mare decât al doilea.

```
public class Currency : Comparable<Currency>
{
    private int dollars, cents;
    public int CompareTo(Currency other)
    {
        int thisValue = (this.dollars * 100) + this.cents;
        int otherValue = (other.dollars * 100) + other.cents;
        return thisValue.CompareTo(otherValue);
    }
}
```

Interfața **IEquatable**

- ▶ Permite implementarea unui mecanism de verificare dacă două instanțe aparținând unui tip sunt egale
- ▶ Versiunea metodei **Object.Equals**
 - ▶ E verificată și compatibilitatea tipurilor

```
public class Product : IEquatable<Product>
{
    public bool Equals(Product other) ...
}
```

Academia Microsoft

Interfața generică **IEquatable** permite codului client să compare instanțe ale unei clase sau structuri pentru cazuri de egalitate. Implementați această interfață pentru clase și structuri, dacă doriți să definiți cod particularizat pentru a testa egalitatea dintre două instanțe ale aceluiași tip.

Interfața generică *IEquatable* definește o singură metodă, **Equals**.

```
public class Product : IEquatable<Product>
{
    private int _id;
    private string _name;
    private double _price;
    public bool Equals(Product other)
    {
        return (this._id == other._id) &&
            (this._name == other._name) &&
            (this._price == other._price);
    }
}
```

Exemplu de utilizare al acestei interfețe:

```
Product p1 = new Product(1, "Chai", 18.00);
Product p2 = new Product(1, "Chai", 18.00);
if (p1.Equals(p2))
    Console.WriteLine("The objects are equal.");
```

*Interfața **ICollectionable**

- ▶ Permite conversia unui tip într-o mulțime de alte tipuri (tipuri primitive de date, string etc.)

```
public class Currency : ICollectionable
{
    public bool ToBoolean(IFormatProvider p) ...
```

*Interfața **ICloneable**

- ▶ Permite clonarea unui obiect
 - ▶ Obiectul clonat este o copie a obiectului original

```
public class PersonalDetails : ICloneable
{
    public object Clone() ...
```

*Interfața **IFormattable**

- ▶ Permite formatarea unui obiect într-un string
 - ▶ Tipurile formatabile definesc lista de formate permise

```
public struct Point : IFormattable
{
    public string ToString(string fmt, IFormatProvider fp) ...
```

Academia Microsoft

Interfața **ICollectionable** definește metode care convertesc valoarea unei clase sau a unei structuri într-o gamă de diferite tipuri de date. Tipurile de date care sunt acceptate sunt *Boolean*, *SByte*, *Byte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64*, *UInt64*, *Single*, *Double*, *Decimal*, *DateTime*, *Char*, și *String*.

Dintre metodele acestei interfețe mai importante sunt *ToBoolean*, *ToSByte*, *ToByte*, *ToInt16*, *ToUInt16*, *ToInt32*, *ToUInt32*, *ToInt64*, *ToUInt64*, *ToDouble*, *ToFloat*, *ToChar*, etc.

În cazul în care nu există nici o conversie care să aibă sens pentru un anumit tip de date, metoda de conversie aruncă excepția *InvalidCastException*. De exemplu, dacă o clasă reprezintă o valoare booleană, metoda *ToString* poate arunca o excepție, deoarece nu există nicio formă de reprezentare *String* pentru valori booleene.

Următoarele exemple de cod arată o implementare parțială a interfeței *ICollectionable*. Clasa *Currency* definește o proprietate ajutor, care este numită *Value* și care întoarce valoarea monedei ca o valoare de tip *Double*. Metodele de conversie utilizează această proprietate împreună cu metodele predefinite din clasa *System.Convert* pentru a efectua operațiunea de conversie corespunzătoare. Rețineți că metoda de conversie *ToString* aruncă o excepție, deoarece nu există nici o conversie care are sens de la o valoare de tip *Currency* la un *DateTime*.

```

public class Currency : IConvertible
{
    private int _dollars, _cents;

    // Indicate that the TypeCode for a Currency object is
    TypeCode.Object.
    public TypeCode GetTypeCode()
    {
        return TypeCode.Object;
    }

    // Helper property, used by the conversion methods.
    private double Value
    {
        get { return _dollars + (_cents/100.0); }
    }

    public bool ToBoolean(IFormatProvider provider)
    {
        if (_dollars != 0 || _cents != 0)
            return true;
        else
            return false;
    }

    // Most of the conversion methods are variations on
    this method.
    public byte ToByte(IFormatProvider provider)
    {
        return Convert.ToByte(Value);
    }

    // No meaningful conversion to DateTime.
    public DateTime ToDateTime(IFormatProvider provider)
    {
        throw new InvalidCastException("Invalid cast
specified.");
    }

    // Return a locale-specific string, formatted as a
    currency
    public string ToString(IFormatProvider provider)
    {
        return String.Format(provider, "{0:c}", Value);
    }

    // Convert the Currency object to the specified type.
    public object ToType(Type conversionType,
    IFormatProvider provider)
    {
        return Convert.ChangeType(Value, conversionType);
    }
}

```


*Interfața **IConvertible**

- ▶ Permite conversia unui tip într-o mulțime de alte tipuri (tipuri primitive de date, string etc.)

```
public class Currency : IConvertible
{
    public bool ToBoolean(IFormatProvider p) ...
}
```

*Interfața **ICloneable**

- ▶ Permite clonarea unui obiect
 - ▶ Obiectul clonat este o copie a obiectului original

```
public class PersonalDetails : ICloneable
{
    public object Clone() ...
}
```

*Interfața **IFormattable**

- ▶ Permite formatarea unui obiect într-un string
 - ▶ Tipurile formatabile definesc lista de formate permise

```
public struct Point : IFormattable
{
    public string ToString(string fmt, IFormatProvider fp) ...
}
```

Academia Microsoft

Interfața **ICloneable** permite codului client să creeze o nouă instanță a unei clase cu aceeași valoare ca și a unei alte instanțe existente. Implementați această interfață pe clase care necesită acest comportament de copiere în întregime, astfel că obiectul clonă să conțină o copie completă a tuturor membrilor din obiectul original.

Dacă doriți doar să obțineți o copie superficială a unui obiect, nu e nevoie să implementați interfața **ICloneable**. În schimb, invocați metoda **MemberwiseClone** pe un obiect. Metoda **MemberwiseClone** copiază toate variabilele instanței și variabilele de clasă de la o instanță la alta. În cazul în care instanța inițială conține referințe către vreun obiect, clona, va conține referințe care se referă la aceleași obiecte, nu doar copii ale acelor obiecte.

Interfața **ICloneable** definește o singură metodă, **Clone**, care crează o copie în întregime a instanței curente. Exemplu de implementare a interfeței:

```
public class PersonalDetails : ICloneable
{
    private string _name;
    private List<string> _contactNumbers;

    public PersonalDetails(string n)
    {
        _name = n;
        _contactNumbers = new List<string>();
    }
    public void AddContactNumber(string number)
    {
        _contactNumbers.Add(number);
    }
}
```

```

public object Clone()
{
    PersonalDetails cloneObject = new
    PersonalDetails(_name);
    foreach (string num in _contactNumbers)
    {
        cloneObject._contactNumbers.Add(num);
    }
    return cloneObject;
}
}

```

Exemplu de utilizare al unei clase ce implementează această interfață:

```

PersonalDetails originalDetails = new
PersonalDetails("Ruby");

originalDetails.AddContactNumber("Number1");
originalDetails.AddContactNumber("Number2");

PersonalDetails cloneDetails =
(PersonalDetails)originalDetails.Clone();

originalDetails.AddContactNumber("Number3"); // Add a
contact to original object.
cloneDetails.AddContactNumber("Number4"); // Add a
contact to clone object.

```

După executarea codului, originalDetails va conține Number1, Number2, Number3, iar cloneDetails Number1, Number2, Number4.

*Interfața **ICollectionable**

- ▶ Permite conversia unui tip într-o mulțime de alte tipuri (tipuri primitive de date, string etc.)

```
public class Currency : ICollectionable
{
    public bool ToBoolean(IFormatProvider p) ...
```

*Interfața **ICloneable**

- ▶ Permite clonarea unui obiect
 - ▶ Obiectul clonat este o copie a obiectului original

```
public class PersonalDetails : ICloneable
{
    public object Clone() ...
```

*Interfața **IFormattable**

- ▶ Permite formatarea unui obiect într-un string
 - ▶ Tipurile formatabile definesc lista de formate permise

```
public struct Point : IFormattable
{
    public string ToString(string fmt, IFormatProvider fp) ...
```

Academia Microsoft

Interfața **IFormattable** permite codului client să formateze valoarea unui obiect într-o reprezentare sub forma unui șir. Implementați această interfață pentru clase sau structuri pe care doriți să le formatați ca șiruri de caractere în metode, cum ar fi *Console.WriteLine* și *String.Format*.

Interfața **IFormattable** definește o singură metodă, **ToString**. Metoda primește două argumente: un format și un obiect *IFormatProvider*.

În exemplul de mai jos, structura *Point*, implementează metoda *ToString* a interfeței *IFormattable* după cum urmează:

- dacă formatul este null atunci va returna coordonatele x și y ale obiectului
- dacă formatul este "x", atunci întoarce coordonata x
- dacă formatul este "y", atunci întoarce coordonata y
- dacă formatul este orice altceva se va arunca o excepție (**FormatException**)

```
public struct Point : IFormattable
{
    private int _x, _y;

    //Override the ToString method from the System.Object
    class

    public override string ToString()
    {
        return ToString(null, null);
    }
}
```

// Implement the ToString method from the IFormattable interface.

```
public String ToString(string format, IFormatProvider
fp)
{
    if (format == null)
        return String.Format("{0}, {1}", _x, _y);

    else if (format == "x")
        return _x.ToString();

    else if (format == "y")
        return _y.ToString();

    else
        throw new FormatException( String.Format
("Invalid format:'{0}'.", format));

}
}
```

Exemplu de utilizare al interfeței:

```
Point p = new Point(10, 20);
Console.WriteLine("This prints the X coordinate: {0:x}",
p);
Console.WriteLine("This prints the Y coordinate: {0:y}",
p);
Console.WriteLine("This causes a FormatException: {0:z}",
p);
```

Interfața **IDisposable**

- O clasă trebuie să asigure managementul resurselor
- Această interfață permite administrarea timpului de viață al unui obiect și eliberarea resurselor utilizate de acesta

```
public class MyResourceHolder : IDisposable
{
    public void Dispose() ...
}
```

Dacă un obiect utilizează un set specific de resurse, poate elibera aceste resurse prin utilizarea unui destructor. GB execută destructorul atunci când distruge obiectul. Cu toate acestea, unele resurse sunt prea valoroase pentru a aștepta un interval arbitrar de timp până când GB le eliberează. Resursele limitate trebuie să fie eliberate în cel mai scurt timp posibil.

Un exemplu de clasă care alocă resursele sistemului unei aplicații este clasa *TextReader* din spațiul de nume *System.IO*. Această clasă oferă un mecanism pentru a citi caractere dintr-un flux primit ca input secvențial. Clasa *TextReader* conține o metodă virtuală, care este numită *Close*, prin care se închide fluxul. Clasa *StreamReader* (care citește caractere dintr-un flux, cum ar fi un fișier deschis) și clasa de *StringReader* (care citește caractere dintr-un șir), ambele provin din *TextReader*, și ambele suprascriu metoda *Close*.

```
TextReader reader = new StreamReader(filename);
string line = reader.ReadLine();

while (line != null)
{
    Console.WriteLine(line);
    line = reader.ReadLine();
}
reader.Close();
```

Codul apelează metoda `ReadLine` pentru a citi linie cu linie din fluxul de text într-o variabilă șir de caractere. Termină când nu mai are text de citit. Este important să fie apelată metoda `Close` la final pentru a închide atât handler-ul fișierului cât și resursa asociată. Totuși există o problemă cu acest exemplu. Dacă metodele `ReadLine` și `WriteLine` vor genera o excepție atunci `Close` nu se va mai executa.

O variantă prin care să ne asigurăm că metoda `Close` este întotdeauna apelată este utilizarea unui bloc `finally` ca în exemplul următor:

```

TextReader reader = new StreamReader(filename);
try
{
    string line = reader.ReadLine();
    while (line != null)
    {
        Console.WriteLine(line);
        line = reader.ReadLine();
    }
}
finally
{
    reader.Close();
}

```

Această abordare este mai bună decât a nu fi pregătiți să ne ocupăm de excepții, dar are mai multe neajunsuri:

- Rapid devine greoi, dacă trebuie să dispunem mai mult de o resursă (ar însemna să avem mai multe blocuri îmbricate *try* și *finally*).
- În unele cazuri, poate fi necesar să modificăm codul.
- Aceasta nu reușește să creeze o abstractizare a soluției. Acest lucru înseamnă că soluția este greu de înțeles și trebuie să repetați codul pretutindeni unde aveți nevoie de această funcționalitate.
- Referința la resursă rămâne în domeniul de aplicare, după blocul *finally*. Aceasta înseamnă că puteți încerca accidental să utilizați resursa după ce a fost eliberată.

Soluția este utilizarea instrucțiunii **using**, care asigură un mecanism mult mai ordonat de eliberare a resurselor.

Important! Să nu confundați instrucțiunea *using* cu directiva *using* care include un anumit spațiu de nume.

Exemplu de utilizare al instrucțiunii *using*:

```

using (TextReader reader = new StreamReader(filename))
{
    string line = reader.ReadLine();
    while (line != null)
    {
        Console.WriteLine(line);
        reader.ReadLine();
    }
}

```

Echivalentul codului de mai sus este prezentat mai jos:

```

TextReader reader = new StreamReader(filename);
try
{
    string line;
    while (line != null)
    {
        Console.WriteLine(line);
        line = reader.ReadLine();
    }
}
finally
{
    if (reader != null)
    {
        ((IDisposable)reader).Dispose();
    }
}

```

Variabilele pe care le declarați în using trebuie să fie de un tip care implementează interfața **IDisposable**. Interfața *IDisposable* definește o singură metodă **Dispose** care eliberează toate resursele utilizate de obiectul curent (*StreamReader* în exemplul de mai sus).

Implementarea Interfeței *IDisposable*

Când scrieți o clasă, ar trebui să scrieți un destructor sau să implementați această interfață. Apelul către destructor se va întâmpla, dar nu se știe cu exactitate când. În schimb, se știe exact când va fi apelată metoda *Dispose*.

```

public class MyResourceHolder : IDisposable
{
    private bool _disposed = false;

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!_disposed)
        {
            if (disposing)
            {
                // Dispose managed resources here.
            }

            // Dispose unmanaged resources here.
            // Set the _disposed flag to prevent subsequent
            // disposals.
            _disposed = true;
        }
    }
}

```

```
// Finalization code.
~MyResourceHolder()
{
    Dispose(false);
}
}
```

În acest cod:

- Flag-ul **_disposed** indică dacă obiectul a fost deja eliberat și împiedică codul client să elimine un obiect mai mult decât o singură dată.
- Metoda publică **Dispose** invocă o metodă supraîncărcată protejată, de asemenea, numită *Dispose*, pentru a elibera resursele obiectului. Parametrul *True* indică faptul că, obiectul este eliberat printr-un apel în mod explicit al metodei *Dispose*, și nu printr-o invocare de către CLR a codului de finalizare pentru obiect. Metoda publică *Dispose* invocă apoi metoda **GC.SuppressFinalize** pentru a elimina obiectul din coadă de finalizare a CLR-ului. Această metodă oprește GB să apeleze destructorul pentru acest obiect, pentru că obiectul a fost acum finalizat.
- Metoda protejată *Dispose* eliberează toate resursele pentru obiect. În cazul în care parametrul de eliminare este adevărat, înseamnă că metoda a fost invocată de metoda publică *Dispose*. În cazul în care parametrul eliminare este fals, înseamnă că metoda a fost invocată de codul de finalizare, al GC.

Definirea unui tip delegat

- ▶ Un tip delegat încapsulează semnătura unei metode
 - ▶ Util pentru *callback methods*, apeluri asincrone de metode și apeluri amânate ale metodelor

```
delegate double MathOperation(double input);
```

- ▶ Instanțierea unui obiect delegat

```
// Explicit syntax:
MathOperation op = new MathOperation(Math.Sin);

// Implicit syntax:
MathOperation op = Math.Sin;
```

Academia Microsoft

Declarația unui delegat specifică o semnătură de metodă care are un anumit set de argumente și un tip de întoarcere. Când definiți un delegat, compilatorul generează un tip care încapsulează semnătura metodei și permite codului client să invoce metode care au acea semnătură specifică.

Pentru a defini un tip delegat, utilizați cuvântul cheie **delegate**. Exemplu:

```
delegate double MathOperation(double input);
delegate void CallbackOperation(string message);
```

După ce ați definit un tip delegat, puteți crea un obiect pe care să îl asociați cu o metodă particulară a unui obiect sau a unei clase.

Pentru a crea un obiect delegat, puteți utiliza operatorul **new** și specificați numele unei metode ca parametru. Exemplu:

```
MathOperation op = new MathOperation(Math.Sin);
```

Versiunea 2.0 și 3.5 a *.NET Framework* vă permite să utilizați o notăție prescurtată pentru a crea obiecte delegat. Aveți posibilitatea să declarați o variabilă delegat și să îi asigurați un nume de metodă direct; nu trebuie să creați un nou obiect delegat în mod explicit. Compilatorul injectează codul pentru a crea obiect delegat implicit.

```
MathOperation op = Math.Sin;
```

Invocarea unei metode utilizând un delegat

- ▶ O metodă poate fi invocată utilizând un obiect delegat
- ▶ Invocarea poate fi sincronă sau asincronă

```
// Synchronous call:
double result = op(Math.PI/2);

// Asynchronous call:
AsyncCallback callback = MyCallbackMethod;
op.BeginInvoke(Math.PI / 2, callback, null);
```

Academia Microsoft

După ce ați creat un obiect delegat care face referire la o metodă specială a unui obiect sau a unei clase, puteți utiliza acest obiect delegat pentru a invoca metoda sincron sau asincron.

Invocarea sincronă a unei metode prin utilizarea unui delegat

Pentru a invoca o metodă sincron cu ajutorul unui delegat, se utilizează numele instanței delegatului, urmat de argumentele care sunt transmise în metodă. În cazul în care metoda returnează o valoare, puteți prelua valoarea ca și cum ați face-o pentru un apel direct al metodei.

Următoarele exemple de cod arată cum să creați o instanță delegat care reprezintă metoda *Math.Sin* și apoi invocă metoda sincron prin utilizarea instanței delegatului.

```
delegate double MathOperation(double input);
...
MathOperation op = new MathOperation(Math.Sin);
double result = op(Math.PI/2);
```

Invocarea asincronă a unei metode prin utilizarea unui delegat

Pentru a invoca o metodă asincron prin utilizarea unei instanțe delegat, apăsați metoda **BeginInvoke** a instanței delegatului. Metoda *BeginInvoke* va invoca metoda target asincron într-un alt thread. Transmiteți următorii parametrii pentru *BeginInvoke* în ordinea în care sunt specificați:

- Parametrii pe care metoda target îi necesită
- Un obiect delegat **System.AsyncCallback** care reprezintă o metodă de callback. CLR invocă metoda specificată de callback atunci când metoda target a finalizat execuția. Dacă nu doriți să ofere o metodă de apel invers, specificați o valoare nulă pentru acest parametru
- Un obiect **System.Object** care transmite informații suplimentare de stare despre metoda target. În cazul în care metoda target a finalizat de execuție, puteți prelua acest obiect pentru a stabili contextul apelului metodei originale.

Pentru a prelua valoarea de returnare, precum și orice parametri de ieșire de la un apel de metodă asincron, apelați metoda **EndInvoke** pe instanță delegat. De obicei, veți apela *EndInvoke* în metoda de callback; la acest punct, știți că metoda target a terminat execuția, precum și că sunteți gata pentru a prelua rezultatele de la metoda target. Exemplu:

```
// Import the namespace for the AsyncResult type.
using System.Runtime.Remoting.Messaging;
...
// Create a delegate object that specifies the callback
method.
AsyncCallback callback = MyCallbackMethod;

// Create a delegate object that specifies the target
method, and then
// invoke the target method asynchronously.
MathOperation op = Math.Sin;
op.BeginInvoke(Math.PI / 2, callback, null);

// Callback method must have the signature shown here.
void MyCallbackMethod(IAsyncResult ar)
{
    // Create a delegate object that represents the target
    method.
    AsyncResult asyncResult = (AsyncResult)ar;
    MathOperation op =
        (MathOperation)asyncResult.AsyncDelegate;

    // Retrieve and display the result of the target
    method.
    double result = op.EndInvoke(ar);
    Console.WriteLine("Result: {0}", result);
}
```

Definirea și utilizarea evenimentelor

- ▶ Evenimentele permit obiectelor să anunțe alte obiecte contractante că ceva s-a întâmplat
 - ▶ Aplicațiile pot avea definite evenimente în structuri, clase sau interfețe
- ▶ Pentru a defini un eveniment:
 - ▶ Se declară evenimentul în structură clasă sau interfață
 - ▶ Se declanșează evenimentul în cadrul obiectului sursă
 - ▶ Se tratează evenimentul în codul client

Un eveniment este o modalitate pentru un obiect de a notifica aplicațiile client atunci când se întâmplă ceva important acestuia. Utilizarea cea mai des întâlnită a evenimentelor este în aplicații grafice (GUI). Clase care reprezintă controale în GUI ridică evenimente atunci când utilizatorul interacționează cu ele. De exemplu, atunci când utilizatorul face click pe un buton, obiectul butonul ridică un eveniment **Click**. Utilizarea de evenimente nu se limitează doar la GUI. Aveți posibilitatea să utilizați evenimente pentru aplicații business, cum ar fi o schimbare a soldului unui cont bancar, sosirea unui mesaj într-o coadă, sau actualizarea unei înregistrări în baza de date.

Pentru a utiliza evenimente într-o cerere .NET Framework, efectuați următoarele sarcini:

- Declarați un eveniment.
- Ridicați evenimentul.
- Tratați evenimentul.

Declarea unui eveniment

Aveți posibilitatea să declarați evenimente într-o clasă, structură, sau de interfață. Pentru a declara un eveniment, utilizați cuvântul cheie **event**. Declarația evenimentului trebuie să precizeze semnătura unor metode care se va ocupa de eveniment; utilizați una dintre următoarele tehnici pentru a specifica semnătura:

- Specificați semnătura în mod explicit, ca parte a declarației de eveniment.
- Specificați semnătura implicit, prin utilizarea unui tip existent delegat. Tipul de delegat indică semnătura metodelor care vor trata evenimentul (handler).

Dupa versiunea 2.0, a fost introdus tipul de delegat generic: **EventHandler<T>**

.NET Framework definește un model standard pentru metodele care tratează evenimentele, după cum urmează:

- Primul parametru este de tip **System.Object** și ar trebui să indice obiectul care a ridicat evenimentul.
- Al doilea parametru este de tip **EventArgs** sau o subclasă și ar trebui să transmită informații contextuale despre eveniment.

Dacă nu doriți să transmiteți informații contextuale pentru un eveniment, puteți folosi tipul delegat nongeneric *EventHandler* pentru evenimentul dumneavoastră. Delegatul nongeneric *EventHandler* specifică faptul că metodele handler trebuie să definească un parametru *System.Object* și un parametru *EventArgs*.

Următoarele exemple de cod arată cum se definesc evenimente într-o clasă *Account* pentru a indica atunci când un cont bancar devine credit sau deficit.

- clasa *AccountEventArgs* este un eveniment dat ca argument customizat care indică soldul curent al unui cont atunci când apare un eveniment.
- clasa *Account* definește două evenimente publice, care sunt numite *AccountInDeficit* și *AccountInCredit*. Ambele evenimente sunt definite ca *EventHandler<AccountEventArgs>* care specifică semnătură pentru metodele handler.

```
// Custom event argument class.
public class AccountEventArgs : EventArgs
{
    private double _balance;
    public AccountEventArgs(double b)
    {
        _balance = b;
    }

    public double Balance
    {
        get { return _balance; }
    }
}

// Account class, defines two public events.
public class Account
{
    public event EventHandler<AccountEventArgs>
        AccountInDeficit;
    public event EventHandler<AccountEventArgs>
        AccountInCredit;
}
```

Ridicarea unui eveniment

Pentru a ridica un eveniment în Visual C#, trebuie mai întâi testat dacă există înregistrate metode handler pentru acest eveniment. Pentru a face acest lucru, verificați că evenimentul nu este null. Dacă evenimentul nu este null, puteți trece la ridicarea evenimentului. Pentru a face aceasta, utilizați numele evenimentului, urmat de lista de argumente pe care doriți să le transmiteți evenimentului între paranteze.

```
public class Account
{
    private double _balance;

    public void Deposit(double amount)
    {
        _balance += amount;

        if (_balance > 0 && _balance <= amount)
        {
            // Just moved into credit, so raise the
            AccountInCredit event.
            // Create a copy of the event object, to
            prevent a race condition.

            EventHandler<AccountEventArgs> handler =
                AccountInCredit;

            if (handler != null)
            {
                AccountEventArgs args = new
                    AccountEventArgs(_balance);
                handler(this, args);
            }
        }
    }
}
```

Tratarea unui eveniment

Pentru a manipula un eveniment, definiți o metodă handler cu semnătura pe care o specifică evenimentul în definiția acestuia, și apoi adăugați evenimentului metoda handler prin utilizarea operatorului += (C# operator). Puteți adăuga chiar mai multe metode handler pentru același eveniment. Atunci când se produce evenimentul, CLR va invoca metodele handler în ordinea în care le-ați adăugat la eveniment. Pentru a elimina o metodă din lista de metode handler pentru un anumit eveniment, folosiți -= .

```
// Create an Account object.
Account acc1 = new Account("Jane");

// Handle the AccountInCredit event on the Account object.
```

```
acc1.AccountInCredit += OnAccountInCredit;

// Deposit and withdraw some money.
acc1.Withdraw(100);
acc1.Deposit(50);
acc1.Deposit(70);

// Event-handler method for the AccountInCredit event.
void OnAccountInCredit(object sender, AccountEventArgs
args)
{
    double balance = args.Balance;
    Console.WriteLine("Account in credit, new balance:
{0:c}", balance);
}
```


 itacad
 you@technology

Aruncarea și prinderea excepțiilor

- ▶ Excepțiile se aruncă atunci când se detectează condițiile unei erori
 - ▶ Excepția este un obiect care descrie o eroare
 - ▶ Toate excepțiile moștenesc **System.Exception**
- ▶ Pentru a ridica o excepție se folosește **throw**
- ▶ Pentru a prinde o excepție se folosesc blocuri **try/catch** și **finally**

 Microsoft
 .NET

Academia Microsoft

O aplicație poate să arunce o excepție atunci când întâlnește o condiție în afara controlului său. În cazul în care cererea aruncă o excepție, CLR caută în arborele de apeluri pentru un bloc **catch** care să manipuleze acest tip de excepție. Pentru a arunca o excepție, se utilizează **throw**, urmat de obiectul excepție. .NET Framework conține un număr mare de clase predefinite pentru excepțiile comune.

```
public void Deposit(double amount)
{
    if (amount <= 0)
        throw new System.ArgumentException("Amount must be a
        positive number.");
    _balance += amount;
}
```

Pentru a prinde excepțiile se folosesc fie blocurile **try/catch** sau **finally**:

```
try
{
    // Some code that may throw an exception.
}
catch (SomeException ex)
{
    // Code to react to the occurrence of the exception.
    Console.WriteLine("Exception occurred: {0}", ex.Message);
}
finally
{
    // Code that runs under any circumstances.
}
```


- Blocul **try** conține cod care poate arunca o excepție. Atunci când o excepție este aruncată în acest bloc, primul bloc *catch* ale cărui filtre se potrivesc cu clasa din care face parte excepția o prinde. Rețineți că filtrul este specificat ca parte a clauzei *catch*.
- Blocul **catch** prinde excepții de un anumit tip. Obiectul excepție oferă informații despre eroare; de exemplu, toate obiectele excepție au o proprietate *Message* care descrie eroarea.
- Blocul **finally** întotdeauna se execută, indiferent dacă este aruncată o excepție. *finally* efectuează operațiuni de curățare, cum ar fi de închidere a conexiunilor într-o bază de date.

Ar trebui să capturați excepții numai atunci când trebuie să efectuați în mod specific una dintre următoarele acțiuni:

- Să păstrați informații cu privire la excepție într-un fișier de log-uri.
- Să adăugați orice informații relevante pentru o excepție.
- Să executați cod de curățare.

Ierarhia de excepții

- ▶ Se pot defini multiple blocuri **catch** pentru a trata diverse erori
 - ▶ Ne permite să lucrăm cu ierarhia de excepții

```
try
{ ... }
catch (FileNotFoundException e)
{ ... }
catch (IOException e)
{ ... }
catch (Exception e)
{ ... }
```

Academia Microsoft

Toate clasele excepție sunt moștenite din clasa **System.Exception**. Clasele excepții predefinite moștenesc *System.Exception*. Puteți, de asemenea, să definiți propriile excepții customizate; în această situație, definiți o clasă excepție care moștenește *System.Exception*.

Se pot defini mai multe blocuri **catch**, secvențial pentru a prinde diferite tipuri de excepție. Atunci când apare o excepție, CLR analizează fiecare bloc *catch* în ordinea în care a fost definit pentru a-l găsi pe cel potrivit care se poate ocupa de excepție. Prin urmare, trebuie să ordonați blocurile *catch* de la cel mai specific până la cel mai generic pentru a vă asigura că CLR execută blocul cel mai specific pentru orice excepție dată.

```
try
{
    // File operations.
}
catch (System.IO.FileNotFoundException ex)
{
    Console.WriteLine("File not found: {0}", ex.FileName);
}
catch (System.IO.IOException ex)
{
    Console.WriteLine("General I/O exception: {0}",
        ex.Message);
}
catch (System.Exception ex)
{
    Console.WriteLine("General exception: {0}",
        ex.Message);
}
```

Definirea excepțiilor custom

- ▶ Puteți defini noi tipuri de excepții
 - ▶ Moșteniți din clasa `Exception` sau orice altă subclasă

```
public class MyException: Exception
{
    public MyException() ..
    public MyException(string message) ..
    public MyException(string message, Exception
        inn) ..
}
```

Puteți defini o ierarhie personalizată de clase excepție specifice pentru o anumită aplicație care moștenesc **System.Exception** și reprezintă condiții specifice de eroare pentru aplicația dată. Există mai multe motive pentru a defini o ierarhie de tipuri de excepție personalizate: Dezvoltarea este simplificată, deoarece puteți defini proprietățile și metodele dvs pentru clasa excepție de bază, iar alte clase excepție din aplicație o pot moșteni.

.NET Framework furnizează o ierarhie extinsă de clase excepție. În cazul în care *.NET* oferă deja o clasă corespunzătoare pentru o anumită situație, folosiți acea clasă excepție și nu definiți una nouă. Definiți o clasă nouă pentru excepții care nu sunt deja disponibile *.NET Framework*.

Când creați ierarhia dvs., utilizați următoarele întrebări pentru a vă ajuta să decideți dacă trebuie să creați o nouă clasă excepție:

- Există deja o excepție pentru această situație? În cazul în care există o excepție, fie în ierarhia dvs. curentă sau în *.NET Framework*, utilizați acea clasă excepție.
- Este o excepție specială, care solicită manipulare discretă? Dacă este așa, creați o clasă excepție pentru a permite codului dvs. să prindă excepția specifică și să o trateze în mod explicit. Aceasta elimină necesitatea de a captura o excepție mai generică și apoi utilizați logica condițională pentru a stabili ce măsuri să fie luate.
- Aveți nevoie de comportament specific sau informații suplimentare pentru o excepție specială? Dacă da, aveți posibilitatea să utilizați o clasă excepție în aplicație pentru a include informații suplimentare sau funcționalitatea pentru a se potrivi cu o cerință specifică.

Pentru a respecta standardul în denumire, întotdeauna folosiți la sfârșitul numelui clasei, cuvântul *Exception*. Clasa dvs. de bază pentru excepții ar trebui să definească domeniiile pentru captarea informațiilor specifice, cum ar fi data și ora când a avut loc excepția, precum și numele computerului.

```
public class YourBaseApplicationException :
ApplicationException
{
    // Default constructor.

    public YourBaseApplicationException()
    {
        ...
    }

    // Constructor that accepts a single string message.

    public YourBaseApplicationException(string message) :
base(message)
    {
        ...
    }

    // Constructor that accepts a string message and an
    inner exception
    // that will be wrapped by the custom exception class.

    public YourBaseApplicationException(string message,
Exception inner)
    : base(message, inner)
    {
        ...
    }
}
```

Best Practice

- ▶ Evitați boxing și unboxing atunci când este posibil
- ▶ Utilizați tipuri *nullable* pentru tipurile de valoare care nu au o valoare nulă predefinită
- ▶ Când definiți un tip generic, definiți și constrângerile asupra acestuia
- ▶ Implementați interfețele standard unde este nevoie
- ▶ Definiți noi clase de excepții cât mai specifice pentru aplicație

Sumar

- ▶ Tipuri valoare și referință
- ▶ Utilizarea tipurilor generice
- ▶ Implementarea interfețelor standard
- ▶ Implementarea delegațiilor și evenimentelor
- ▶ Excepții