





# Modulul 10

## Proprietăți și indecși




itacad  
you@technology



## Overview

- ▶ Utilizarea proprietăților
- ▶ Utilizarea indecșilor



2

- Atributele unei clase pot fi accesate prin câmpuri sau proprietăți. Câmpurile sunt implementate ca variabile membre **private** ale clasei. În C# proprietățile apar a fi câmpuri pentru utilizator, dar folosesc metode pentru a obține sau pentru a seta valori.
- C# mai pune la dispoziție indecșii, care ne permit să indexăm un obiect ca și cum ar fi un vector.
- După completarea acestui modul studentul va putea să:
  - Creeze proprietăți care încapsulează date în cadrul unei clase
  - Definească indecși prin care să acceseze o clasă utilizând notațiile folosite la vectori


itacad  
you@technology

Microsoft  
.NET

## De ce să folosim proprietăți?

- ▶ Proprietățile asigură
  - ▶ O modalitate utilă de a încapsula informație într-o clasă
  - ▶ Sintaxă concisă
  - ▶ Flexibilitate

3

➤ Proprietățile oferă o modalitate utilă de a încapsula date într-o clasă. Exemple de proprietăți includ: lungimea unui șir de caractere, mărimea unui font, numele unui client etc.

### ➤ Sintaxă concisă

▪ Proprietățile sunt elemente de bază ale limbajului. Gândiți-vă la o proprietate ca la un câmp care vă ajută să vă concentrați pe logica aplicației. Să urmărim exemplele de mai jos, primul fără proprietate, al doilea cu proprietate:

```
o.SetValue(o.GetValue( ) + 1);
```

```
o.Value++;
```

▪ Instrucțiunea care folosește o proprietate este mult mai ușor de înțeles și mai puțin susceptibilă la erori.

### ➤ Flexibilitate

▪ Pentru a scrie și a citi valoarea unui câmp se folosesc metode de accesare. Compilatorul translatează sintaxa acestor pseudo câmpuri în accesorii **get** și **set**. Orice modificare realizată în corpul accesoriilor **get** și **set** nu afectează în niciun fel folosirea proprietății (din perspectiva utilizatorului nu s-a produs nicio schimbare). Datorită acestei separări, ce oferă o flexibilitate evidentă pentru dezvoltator, este recomandat să se folosească, pe cât posibil, proprietățile în favoarea câmpurilor.



## Accesarea câmpurilor: **get** / **set**

### ► Proprietățile oferă modalități de a accesa câmpurile

- **get** – folosit pentru citirea unui câmp
- **set** – folosit pentru scrierea unui câmp

```
class Button
{
    public string Caption // Property
    {
        get { return caption; }
        set { caption = value; }
    }
    private string caption; // Field
}
```

4

► **Proprietatea** este un membru al clasei care ne permite să accesăm câmpurile unui obiect. Asociem proprietatea cu cele două acțiuni care se pot executa asupra câmpurilor unui obiect: scriere și citire. Declararea unei proprietăți presupune specificarea numelui acesteia, tipul și două porțiuni de cod cunoscute ca accesori: **get** și **set**.

► Accesorii nu au parametri. O proprietate nu trebuie neapărat să aibă și **get** și **set**. De exemplu o proprietatea *read-only* are doar accesorul **get**.

#### ► **get**

- Returnează valoarea unei proprietăți:

```
public string Caption
{
    get { return caption; }
    ...
}
```

- Implicit este chemat accesorul **get** al proprietății atunci când aceasta este folosită într-un context de citire:

```
Button myButton;
...
string cap = myButton.Caption; // Calls "myButton.Caption.get"
```

- Observați că nu se folosesc paranteze după numele proprietății. În acest exemplu instrucțiunea **return caption** întoarce un șir de caractere. Acest *string* este returnat oricând valoarea proprietății **Caption** este citită.

#### ► **set**

- Modifică valoarea unei proprietăți:

```
public string Caption
{
    ...
    set { caption = value; }
}
```

- Implicit este apelat accesorul **set** al proprietății atunci când proprietatea este folosită în contextul unei scrieri – când e folosită într-o atribuire:

**Button myButton;**

...

**myButton.Caption = "OK"; // Calls "myButton.Caption.set"**

- Observați din nou că nu se folosesc paranteze. Variabila **value** conține valoarea care va fi atribuită proprietății și va fi creată de compilator. În interiorul accesoriului **set** a proprietății **Caption**, valoarea poate fi văzută ca un *string* care conține "OK". **set** nu poate returna o valoare.
- Invocarea unui accesoriu **set** este sintactic identic cu o atribuire, deci ar trebui să limitați efectele laterale observabile. De exemplu ar fi ceva de neașteptat pentru următoarea instrucțiune să schimbe și culoarea și viteza obiectului **thing**:  
**thing.speed = 5;**
- Cu toate acestea, efectele laterale sunt benefice uneori. De exemplu, un coș de cumpărături își poate schimba suma de fiecare dată când un nou lucru este introdus în el.


itacad  
you@technology

Microsoft  
.NET

## Comparație proprietate / câmp

- ▶ Proprietatea este un “câmp logic”
  - ▶ Accesorul **get** poate returna o valoare compusă
- ▶ Asemănări
  - ▶ Sintaxa pentru creare și folosire este aceeași
- ▶ Diferențe
  - ▶ Proprietățile nu sunt valori; ele nu au nicio adresă
  - ▶ Proprietățile nu pot fi folosite ca parametrii **ref** sau **out**

5

### ➤ Proprietățile sunt câmpuri logice

- **get** poate fi folosit pentru a calcula o anumită valoare, ceea ce este preferabil față de a trimite direct valoarea câmpului. Proprietățile pot fi considerate câmpuri logice care nu au neapărat o implementare fizică directă. De exemplu o clasă **Person** poate conține un câmp pentru data de naștere și o proprietate pentru vârstă care calculează vârsta pe baza datei de naștere.

```
class Person
{
    public Person(DateTime born)
    {
        this.born = born;
    }
    public int Age
    {
        // Simplified...
        get { return DateTime.UtcNow.Year – born.Year; }
    }
    ...
    private readonly DateTime born;
}
```

### ➤ Asemănări

- Proprietățile sunt o extensie naturală a câmpurilor. La fel ca și câmpurile ele:
  - Se declară specificând numele și tipul care e diferit de **void**:

```
class Example
{
    int field;
    int Property { ... }
}
```

- Pot fi declarate cu orice modificador de acces:

```
class Example
{
    private int field;
    public int Property { ... }
}
```

- Pot fi statice:

```
class Example
{
    static private int field;
    static public int Property { ... }
}
```

- Pot ascunde membrii ai clasei de bază cu același nume:

```
class Base
{
    public int field;
    public int Property { ... }
}
class Example: Base
{
    public int field;
    new public int Property { ... }
}
```

- Folosesc aceeași sintaxă pentru atribuire sau în contextul citirii:

```
Example o = new Example( );
o.field = 42;
o.Property = 42;
```

### ➤ Diferențe față de câmpuri

- Spre deosebire de câmpuri proprietățile nu corespund direct cu o locație de memorie. Chiar dacă este folosită aceeași sintaxă de accesare ca și la câmp, proprietatea nu este o variabilă. Prin urmare nu poți transmite o proprietate ca un parametru **ref** sau **out** fără să primești o eroare de compilare. Exemplu:

```
class Example
{
    public string Property
    {
        get { ... }
        set { ... }
    }
    public string Field;
}
class Test
{
    static void Main( )
    {
        Example eg = new Example( );
        ByRef(ref eg.Property); // Compile-time error
        ByOut(out eg.Property); // Compile-time error
        ByRef(ref eg.Field); // Okay
        ByOut(out eg.Field); // Okay
    }
    static void ByRef(ref string name) { ... }
    static void ByOut(out string name) { ... }
}
```

## Comparație proprietate / metodă

### ► Asemănări

- Conțin cod executabil
- Pot fi folosite pentru a ascunde detalii de implementare
- Pot fi abstracte, virtuale sau suprascrise

### ► Deosebiri

- Sintactic – proprietățile nu au paranteze
- Semantic – proprietățile nu pot fi **void** sau să aibă parametrii

6

### ► Asemănările dinte metode și proprietăți

- Conțin instrucțiuni care vor fi executate
- Este specificat un tip al valorii returnate care are același grad de accesibilitate ca și proprietatea în sine
- Pot fi marcate ca virtuale, supraîncărcate sau abstracte
- Pot fi introduse într-o interfață
- Oferă o separare între structura internă a unui obiect și interfața sa publică
- Acest ultim punct este probabil cel mai important. Poți schimba implementarea unei proprietăți fără ca modificările să afecteze sintaxa de folosire a acesteia. De exemplu, în codul următor observați că proprietatea **TopLeft** a clasei **Label** este implementată direct, cu câmpul **Field**:

```
struct Point{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int x, y;
}
class Label
{
    ...
    public Point TopLeft
    {
        get { return topLeft; }
        set { topLeft = value; }
    }
    private Point topLeft;
}
```



```
class Use
{
    static void Main( )
    {
        Label text = new Label(...);
        Point oldPosition = text.TopLeft;
        Point newPosition = new Point(10,10);
        text.TopLeft = newPosition;
    }
    ...
}
```

- Pentru că **TopLeft** este o proprietate, ea poate fi modificată fără a afecta sintaxa de folosire, așa cum este prezentat în exemplul următor:

```
class Label
{
    public Point TopLeft
    {
        get { return new Point(x,y); }
        set { x = value.x; y = value.y; }
    }
    private int x, y;
}
class Use
{
    static void Main( )
    {
        Label text = new Label(...);
        // Exactly the same
        Point oldPosition = text.TopLeft;
        Point newPosition = new Point(10,10);
        text.TopLeft = newPosition;
        ...
    }
}
```

### ➤ Deosebiri

- Principalele deosebiri dintre metode și proprietăți sunt sumarizate în tabelul următor:

Caracteristici	Proprietăți	Metode
Folosirea parantezelor	Nu	Da
Parametrii	Nu	Da
Utilizarea tipului <b>void</b>	Nu	Da

➤ Considerăm următoarele exemple pentru a evidenția deosebirile:

- Folosirea parantezelor:

```
class Example
{
    public int Property { ... }
    public int Method( ) { ... }
}
```

- Existența parametrilor:

```
class Example
{
    public int Property { ... }
    public int Method(double d1, decimal d2) { ... }
}
```

- Utilizarea tipului **void**:

```
class Example
{
    public void Property { ... } // Compile-time error
    public void Method( ) { ... } // Okay
}
```

## Tipuri de proprietăți

- ▶ Proprietăți *read / write*
  - ▶ Au ambii accesorii: **get** și **set**
- ▶ Proprietăți *read-only*
  - ▶ Au doar accesoriul **get**
  - ▶ Nu pot fi constante
- ▶ Proprietăți *write-only* (utilizare limitată)
  - ▶ Au doar accesoriul **set**
- ▶ Proprietăți statice
  - ▶ Au ca referință clase
  - ▶ Accesează doar date statice

7

➤ Când folosim proprietățile, putem să specificăm ce operații sunt permise pentru fiecare proprietate în parte:

- **read/write** când implementezi ambii accesorii **get** și **set**
- **read** când implementezi numai accesoriul **get**
- **write** când implementezi numai accesoriul **set**

➤ **Proprietățile read-only**

- Proprietățile care au implementat numai accesoriul **get** se numesc **read-only**. În exemplul de mai jos, clasa **BankAccount** are o proprietate **Balance** care are accesoriul **get**, dar nu și **set**:

```
class BankAccount
{
    private decimal balance;
    public decimal Balance
    {
        get { return balance; } // But no set
    }
}
```

- Nu poți atribui o valoare unei astfel de proprietăți. De exemplu, dacă executăm instrucțiunea de mai jos pentru exemplul anterior vom primi o eroare de compilare:

```
BankAccount acc = new BankAccount();
acc.Balance = 1000000M; // Compile-time error
```

- O eroare foarte comună este să considerăm că o proprietate **read-only** specifică o valoare constantă. Nu este cazul. În exemplul următor, **Balance** este o proprietate **read-only** ceea ce înseamnă că poate fi doar citită. Totuși, valoarea proprietății **Balance** se poate modifica în timp. De exemplu, **Balance** crește atunci când este realizat un depozit:

```
class BankAccount
{
    private decimal balance;
    public decimal Balance
    {
        get { return balance; }
    }
    public void Deposit(decimal amount)
    {
        balance += amount;
    }
    ...
}
```

#### ➤ Proprietăți write-only

- Proprietățile care au doar accesul **set** se numesc **write-only**. În general, ar trebui evitată utilizarea acestora.
- Dacă se va încerca citirea unei proprietăți care este **write-only** se va primi o eroare de compilare.

#### ➤ Proprietăți statice

- O proprietate statică, la fel ca și o metodă statică sau un câmp static, este asociată cu clasa și nu cu obiectul. Din cauza aceasta, trebuie să acceseze numai date statice și nu poate folosi **this**. Codul următor exemplifică aceste aspecte:

```
class MyClass
{
    private int MyData = 0;
    public static int ClassData
    {
        get {
            return this.MyData; // Compile-time error
        }
    }
}
```

- Nu poate fi inclus niciunul dintre modificatorii **virtual**, **abstract** sau **override** în definirea unei proprietăți statice.


itacad  
you@technology

## Un exemplu de proprietate

```
public class Console
{
    public static TextReader In
    {
        get {
            if (reader == null) {
                reader = new StreamReader(...);
            }
            return reader;
        }
        ...
        private static TextReader reader = null;
    }
}
```

8

➤ Proprietățile se pot folosi pentru a întârzia inițializarea unei resurse până în momentul în care este prima oară referită. Această tehnică se numește **lazy creation**, **lazy instantiation** sau **just-in-time creation**. Următorul cod arată un exemplu al acestui concept din *Microsoft .NET Framework SDK*:

```
public class Console
{
    public static TextReader In
    {
        get {
            if (reader == null) {
                reader = new StreamReader(...);
            }
            return reader;
        }
        ...
        private static TextReader reader = null;
    }
}
```

➤ Observații:

- Câmpul **reader** este inițializat cu **null**.
- Numai prima încercare de citire va executa codul din interiorul instrucțiunii **if** creând un obiect **StreamReader**.



## Ce sunt indecșii?

- ▶ Indecșii accesează obiectul ca pe un vector
  - ▶ Util dacă o proprietate are mai multe valori
- ▶ Pentru a defini un index:
  - ▶ Creați o proprietate numită **this**
  - ▶ Specificați tipul indexului
- ▶ Pentru a folosi un index:
  - ▶ Utilizați notațiile unei vector pentru a citi și scrie proprietatea indexată

9

➤ Un obiect este compus din mai multe articole (De exemplu o listă poate fi compusă din mai multe *string*-uri). Indecșii te ajută să accesezi aceste articole utilizând notațiile de la vectori.

### ➤ Definirea unui index

- Următorul cod arată cum să creezi un index pentru a accesa o listă privată de șiruri de caractere

```
class StringList
{
    private string[ ] list;
    public string this[int index]
    {
        get { return list[index]; }
        set { list[index] = value; }
    }
    // Other code and constructors to initialize list
}
```

- Indexul este o proprietate numită **this** și este urmat de paranteze pătrate între care se specifică tipul de index folosit (indecșii trebuie numiți întotdeauna **this**, ei nu au propriile denumiri. Ei sunt accesați din perspectiva obiectului căruia aparțin).

### ➤ Utilizarea unui index

- Putem folosi indexul clasei **StringList** pentru a avea acces la membrii listei **myList** ca în exemplul următor:

```
StringList myList = new StringList( );
myList[3] = "o"; // Indexer write
string myString = myList[3]; // Indexer read
```

- Observați că sintaxa este la fel ca la vectori. Întregul din paranteze specifică ce element este accesat, după care, în funcție de situație (citire sau scriere) se folosesc **get** sau **set**.

## Comparație index / vector

- ▶ Asemănări
  - ▶ Ambele folosesc notațiile de la vector
- ▶ Deosebiri
  - ▶ Pot avea cheie de indexare *non-integer*
  - ▶ Indecșii pot fi supraîncărcați
  - ▶ Indecșii nu sunt variabile
  - ▶ Indecșii nu pot fi parametrii **ref** sau **out**

10

### ➤ Definirea tipului unui index

- Tipul unui index folosit pentru a accesa un vector trebuie să fie **integer**. Indecșii, în schimb, pot avea și alte tipuri. Exemplul următor arată o cheie de indexare de tip **string**:

```
class NickNames
{
    private Hashtable names = new Hashtable( );
    public string this[string realName]
    {
        get { return (string)names[realName]; }
        set { names[realName] = value; }
    }
    ...
}
```

- În exemplul de mai jos, clasa **NickNames** păstrează perechi nume real, poreclă:

```
NickNames myNames = new NickNames( );
myNames["John"] = "Cuddles";
string myNickName = myNames["John"];
```

### ➤ Supraîncărcarea

- O clasă poate avea indecși multipli, dacă ei au tipuri diferite. Poți extinde clasa **NickNames** pentru a crea un index de tip **integer**. Indexul poate itera printr-o tabelă de dispersie (*hashtable*) de un anumit număr de ori și să returneze valoarea corespunzătoare:

```
class NickNames
{
    private Hashtable names = new Hashtable( );
```

```

public string this[string realName]
{
    get { return (string)names[realName]; }
    set { names[realName] = value; }
}
public string this[int nameNumber]
{
    get
    {
        string nameFound;
        // Code that iterates through the Hashtable
        // and populates nameFound
        return nameFound;
    }
}
...
}

```

### ➤ Indecșii nu sunt variabile

- Spre deosebire de vectori, indecșii nu corepund unor locații de memorie. În schimb, ei au accesorii **get** și **set** pentru a executa comenzile de citire și scriere a valorilor. Aceasta înseamnă că, deși ei folosesc aceeași sintaxă ca vectorii, indecșii nu sunt clasificați drept variabile.
- Dacă un index este folosit pe post de parametru **ref** sau **out** se va primi o eroare de compilare ca în exemplul următor:

```

class Example
{
    public string[ ] array;
    public string this[int index]
    {
        get { ... }
        set { ... }
    }
}
class Test
{
    static void Main( )
    {
        Example eg = new Example( );
        ByRef(ref eg[0]); // Compile-time error
        ByOut(out eg[0]); // Compile-time error
        ByRef(ref eg.array[0]); // Okay
        ByOut(out eg.array[0]); // Okay
    }
    static void ByRef(ref string name) { ... }
    static void ByOut(out string name) { ... }
}

```



## Comparație index / proprietate

- ▶ Asemănări
  - ▶ Ambii folosesc **get** și **set**
  - ▶ Nu au adrese
  - ▶ Nu pot fi **void**
- ▶ Deosebiri
  - ▶ Indecșii pot fi supraîncărcați
  - ▶ Indecșii nu pot fi statici

11

➤ Indecșii sunt bazați pe proprietăți, de aceea au foarte multe trăsături în comun cu acestea. Indecșii diferă de proprietăți prin câteva caracteristici. Pentru a înțelege cu adevărat indecșii este binevenită o comparație cu proprietățile.

### ➤ Asemănări

- Ambele folosesc accesorii **get** și **set**
- Nu au adrese de memorie, de aceea nu pot fi parametrii **ref** sau **out**.

```
class Dictionary
{
    public string this[string index]
    {
        get { ... }
        set { ... }
    }
}
Dictionary oed = new Dictionary( );
...
ByRef(ref oed["life"]); // Compile-time error
ByOut(out oed["life"]); // Compile-time error
```

- Nu pot avea tipul **void**

### ➤ Diferențe

- **Identificarea.** O proprietate este identificată doar prin numele său; un index este specificat prin semnatura sa (nume, paranteze pătrate și tipul parametrului după care se indexează).
- **Supraîncărcarea.** O proprietate nu poate fi supraîncărcată, în schimb, un index da(apare și tipul între paranteze pătrate)
- **Static sau dinamic.** O proprietate poate să fie un membru static, pe când un index este întotdeauna un membru instanțiat.

## Utilizarea parametrilor în definirea indecșilor

- ▶ Când definiți un index
  - ▶ Specificați cel puțin un parametru de indexat
  - ▶ Specificați o valoare pentru fiecare parametru
  - ▶ Nu folosiți **ref** sau **out** pentru parametrii

118

- Sunt trei reguli care trebuie urmărite în utilizarea indecșilor:
  - Să se specifice cel puțin un parametru după care se indexează.
  - Să se specifice o valoare pentru fiecare parametru.
  - Să nu se folosească modificatorii **ref** sau **out** pentru parametrii

```
class BadParameter
{
    // Compile-time error
    public string this[ref int index] { ... }
    public string this[out string index] { ... }
}
```

### ➤ Parametrii multipli

- Se poate specifica mai mult de un singur parametru pentru un index.
- Exemplu:

```
class MultipleParameters
{
    public string this[int one, int two]
    {
        get { ... }
        set { ... }
    }
}
```

- Pentru a folosi indexul clasei **MultipleParameters** trebuie specificate două valori:

```
MultipleParameters mp = new MultipleParameters( );
string s = mp[2,3];
```

- Acesta este echivalentul matricilor multidimensionale


itacad  
you@technology

Microsoft  
.NET

## Un exemplu de index: clasa **String**

### ► Clasa **String**

- Este o clasă stabilă
- Folosește un index (**get** dar nu și **set**)

```
class String
{
    public char this[int index]
    {
        get {
            if (index < 0 || index >= Length)
                throw new IndexOutOfRangeException( );
            ...
        }
        ...
    }
}
```

13

### ► Clasa **String**

- Când apelezi o metodă pentru un obiect *String*, este garantat faptul că metoda nu va schimba acel obiect. Dacă metoda returnează un *string* atunci este vorba de un nou șir de caractere (va crea unul nou).

### ► Metoda **Trim()**

- Pentru a înlătura spațiile albe în plus din șirul de caractere se folosește metoda **Trim()**.

```
public sealed class String
{
    public String Trim( ) { ... }
}
```

- Metoda returnează un nou șir de caractere, dar *string*-ul inițial folosit pentru apelare rămâne intact. Exemplu:

```
string s = " Trim me ";
string t = s.Trim( );
Console.WriteLine(s); // Writes " Trim me "
Console.WriteLine(t); // Writes "Trim me"
```

### ► Indexul clasei **String**

- Din cauza faptului că orice metodă care returnează un *string* crează unul nou și îl păstrează intact pe cel inițial, indexul clasei *String* este declarat cu un accesoriu **get** dar fără **set** ca în exemplul următor:

```
class String
{
    public char this[int index]
    {
        get {
            if (index < 0 || index >= Length)
                throw new IndexOutOfRangeException( );
            ...
        }
    }
    ...
}
```

- Dacă se încearcă scrierea utilizând un index se va primi o eroare de compilare:

```
string s = "Sharp";
Console.WriteLine(s[0]); // Okay
s[0] = 'S'; // Compile-time error
s[4] = 'k'; // Compile-time error
```

- Clasa *String* are o clasă pereche numită ***StringBuilder*** care are un index *read/write*.



- Utilizarea proprietăților
- Utilizarea indecșilor



14

- Declarați o clasă **Font** care să aibă o proprietate **Name** *read-only* de tip **String**.
- Declarați o clasă **DialogBox** care să aibă o proprietate **Caption** *read-write* de tip **String**.
- Declarați o clasă **MutableString** care conține un index *read-write* de tip **char** care are un singur parametru de tip **int**.
- Declarați o clasă **Graph** care conține un index *read-only* de tip **double** care are un singur parametru de tip **Point**.