

Programarea Aplicațiilor Windows – curs 2

Prof. dr. Cristian CIUREA
Departamentul de Informatică și Cibernetică Economică
Academia de Studii Economice București
cristian.ciurea@ie.ase.ro

Agenda

1. Clase în C# (derivare/moștenire)
2. Supraîncărcare operatori
3. Proprietăți
4. Interfețe (ICloneable/IComparable)
5. Clase abstracte

Clase

Sintaxă definire:

```
[tip_acces]           [attribute]           class
Nume_Clasa [:clasa_baza, interfata_1,
..., interfata_n]
{
    tip_acces atribut1;
    tip_acces atribut2;
    ...
    tip_acces functie_membral;
};
```

Clase

tip_acces:

- `public` – accesibile;
- `private` – inaccesibile;
- `protected` – accesibile claselor derivate;
- `internal` – accesibile claselor din același *assembly*;
- `protected internal` – accesibile claselor derivate din același *assembly*;

Clase

Atribute constante:

- definite prin **const** sau **readonly**;
- NU este permisă modificarea valorii odată ce au fost inițializate;
- sintaxa:

```
class Test  
{
```

```
    public const int atribut_1 = 10;  
    public readonly int atribut_2;  
}
```

Clase

Atribute constante definite prin **const**:

- sunt inițializate doar la definire;
- NU este permisă modificarea valorii odată ce au fost inițializate;
- sunt considerate variabile definite la nivelul clasei și nu atribute ale unui obiect (**static**);
- pot fi accesate prin numele clasei.

Clase

Atribute constante definite prin **readonly**:

- sunt inițializate la definire sau în constructor;
- NU este permisă modificarea valorii odată ce au fost inițializate;
- reprezintă forma echivalentă a variabilelor **const** din C++.

Clase

Atribute statice definite prin **static**:

- definesc atribute ce nu aparțin unui obiect și pot fi **readonly**;
- inițializarea se face la definire sau prin constructor static;
- sunt considerate variabile definite la nivelul clasei și nu atribute ale unui obiect (**static**);
- sunt accesate prin numele clasei.

Clase

Atribute statice – sintaxă:

```
class Test
{
    public static int atribut_1 = 10;
    public static readonly int atribut_2;
}
```

Clase

Funcții constructor:

- sintaxa:

```
class Nume_clasa {  
    public Nume_clasa ( ) {...}  
};
```

- apel: deoarece obiectele sunt gestionate prin referințe, crearea unui obiect se face cu operatorul *new*:

```
public static void Main ( ) {  
    Nume_clasa obiect_1 = new Nume_clasa ( ) ;  
    Nume_clasa obiect_2 = new Nume_clasa  
        (parametrii constructor)  
}
```

Clase

Funcții destructor:

- sintaxa:

```
class Nume_clasa {  
    ~Nume_clasa( ) {...}  
};
```

- rol principal: eliberarea resurselor gestionate de un obiect (ex: închidere conexiune bază de date, închidere fișier, etc);
- sunt **apelate implicit** de către GC;
- NU pot fi apelate explicit;
- pot fi înlocuite cu metoda **Dispose()** (se poate apela explicit).

Operatori

Supraîncărcare operatori:

- sunt implementați prin funcții statice;
- sunt funcții care se numesc:
operator [simbol]

Operatori

operator =

- rol principal: **copiază bit cu bit** valoarea zonei de memorie sursă în zona de memorie a destinației (cele două zone sunt identice ca structură și tip);
- în cazul obiectelor C#, **copiază valoarea referinței** obiectului sursă în **referința** obiectului destinație.

Operatori

Apel explicit **operator =**

```
class Nume_clasa {  
    ...  
};  
  
public static void Main () {  
    Nume_clasa obiect_1 = new  
        Nume_clasa();  
    Nume_clasa obiect_2 (...) = new  
        Nume_clasa();  
    object_2 = object_1;  
}
```

Operatori

Restricții supraîncărcare operatori:

- NU schimbă precedența operatorilor;
- NU schimbă asociativitatea;
- conservă cardinalitatea (numărul parametrilor);
- NU creează operatori noi;
- formele supraîncărcate nu se compun automat;
- NU se supraîncarcă `=` `.` `?:` `->` `new` `is` `sizeof` `typeof` `[]` `()` `+=` `-=`
- `+=` este evaluat prin operatorul `+`

Operatori

Supraîncărcare operatori unari **++** și **--**:

- 2 forme: prefixată și postfixată;
- prin funcție membră statică;
- cele 2 forme (post și pre) sunt tratate unitar de către compilatorul de C# pentru că lucrăm cu referințe.

Operatori

Supraîncărcare operatori binari **+**, **-**, *****, **/**:

- au întotdeauna 2 parametri;
- comutativitatea operației matematice nu are sens în C# (trebuie definită explicit);
- prin funcție statică publică.

Operatori

Supraîncărcare operator **cast**:

- are întotdeauna 1 parametru;
- numele cast-ului reprezintă tipul returnat;
- NU are tip returnat explicit;
- prin funcție statică;
- folosit la conversia între diferite tipuri de date;
- în C# operatorul are 2 forme de supraîncărcat: **explicit** sau **implicit**;

Operatori

Supraîncărcare operator [] (indexer):

- este o proprietate (se folosește "get");
- pentru că nu are nume se notează cu *this*;
- este folosit pentru a permite acces în citire/scriere pe elementele unui șir de valori din zona privată a obiectului;
- indexul nu este obligatoriu de tip numeric.

Proprietăți

- Proprietățile combină un câmp cu metodele lui de acces, separând accesul în citire de cel de scriere.
- Proprietățile sunt întotdeauna publice și simplifică adresarea câmpurilor private.

Proprietăți

- O proprietate se comportă ca o variabilă care ascunde apeluri de accesorii "get" și "set".
- O clasă poate conține numai unul sau ambii accesorii pentru un câmp.
- Accesorul "set" primește implicit un argument "value" care conține valoarea folosită pentru modificarea câmpului privat.

Proprietăți

- sintaxa:

```
public tip_atribut nume_proprietate {  
    get { return ... }  
    set { }  
}
```

- utilizare:

```
ob1.nume_proprietate = valoare;  
valoare = ob1.nume_proprietate;
```

Proprietăți

```
private int stoc;
```

```
public int Stoc
```

```
{
```

```
    get { return stoc; }
```

```
    set { if (value >= 0) stoc = value; }
```

```
}
```

Interfețe

- Interfețele separă implementarea unui obiect de funcționalitatea lui, adică separă structura unui obiect de modul în care acesta este folosit.
- Interfețele sunt un fel de clase, care conțin doar prototipuri de funcții. Convențional, numele lor începe cu "I".

Interfețe

- Interfețele NU pot fi instanțiate, deoarece conțin doar prototipuri de metode.
- Interfețele NU conțin câmpuri, ci doar metode și, eventual, proprietăți.
- Interfețele NU dispun de constructori și destructor, nefiind niciodată instanțiate.

Interfețe

- clase ce conțin numai funcții abstracte;
- rol de interfață pentru clase care trebuie să definească o serie de metode comune;
- un contract între proprietarii mai multor clase prin care se impune definirea unor serii de metode comune;
- contractul se încheie prin derivarea din interfață;
- se definesc prin **interface** (înlocuiește class).

Interfețe

```
interface IOperatii {  
    void Operatie1( );  
    void Operatie2( );  
};
```

```
class Baza : IOperatii {  
    public void Operatie1() {...}  
    public void Operatie2() {...}  
}
```

Interfața ICloneable

- suportă copierea/clonarea obiectelor unei clase prin crearea unei noi instanțe a respectivei clase, cu aceeași valoare ca o instanță existentă;
- este inclusă în namespace-ul System;
- conține un singur membru, respectiv metoda **Clone()**, care creează un obiect nou, reprezentând copia instanței curente.

Interfața ICloneable

```
public Animal Clone()  
{  
    return (Animal) ((ICloneable) this).Clone();  
    //creeaza o copie a obiectului curent de tip  
    Animal  
}  
  
//metoda Clone() standard/implicita  
object ICloneable.Clone()  
{  
    return this.MemberwiseClone(); //creeaza o  
    copie superficiala a obiectului curent  
}
```

Shallow copy vs. Deep copy

Shallow copy:

- copiere de adrese între 2 obiecte
- realizată implicit prin operatorul =

Deep copy:

- copiere de conținut între 2 obiecte
- realizată explicit prin metode specializate (constructor copiere, Clone(), etc.)

Shallow copy vs. Deep copy

```
Auto a1;  
Auto a2 = new Auto();
```

STIVA

HEAP

referinta null

referinta

X octeti: valori

```
a1 = a2
```

Shallow copy

referinta

referinta

X octeti: valori

```
a1 = new Auto(a2)
```

Deep copy

```
a1 = (Auto)a2.Clone()
```

referinta

referinta

X octeti: valori

X octeti: valori

referinta

referinta

X octeti: valori

X octeti: valori

Interfața IComparable

- definește o metodă generalizată de comparare, pe care o clasă o implementează pentru a-și ordona sau sorta obiectele;
- este inclusă în namespace-ul System;
- conține metoda **CompareTo()**, care compară instanța curentă cu un alt obiect de același tip și returnează un întreg care indică dacă instanța curentă precede, urmează sau se află pe aceeași poziție cu celălalt obiect.

Interfața IComparable

A.CompareTo(B) poate fi:

- $< 0 \Rightarrow$ A precede B;
- $> 0 \Rightarrow$ A urmează B;
- $= 0 \Rightarrow$ A și B se află pe aceeași poziție.

Clase abstracte

- Clasele abstracte se definesc doar în scopul de a deriva din ele alte clase.
- O clasă abstractă nu poate fi instanțiată.
- Clasele abstracte pot să conțină metode abstracte, care nu au implementare în clasa abstractă, ci în clasa derivată.
- O clasă abstractă presupune că un obiect al clasei nu poate fi instanțiat, dar se pot realiza derivări ale acestuia.

Clase sealed

Clase sealed (închise):

- NU este permisă derivarea claselor sealed;

```
sealed class Baza {  
    int atribut1;  
    ...  
};  
class Derivata : Baza { }
```

Clase abstracte vs. Interfețe

Clase abstracte:

- conțin metode abstracte + attribute + metode non-abstracte;
- o clasă poate deriva doar o clasă de bază (abstractă sau nu);
- poate fi utilizată ca reference type.

Interfețe:

- conțin doar metode abstracte;
- o clasă poate deriva mai multe interfețe;
- poate fi utilizată ca reference type.

Bibliografie

- [1] I. Smeureanu, M. Dârdală, A. Reveiu – *Visual C# .NET*, Editura CISON, București, 2004.
- [2] C. Petzold – *Programming Microsoft Windows with C#*, Microsoft Press, 2002.
- [3] L. O'Brien, B. Eckel – *Thinking in C#*, Prentice Hall.
- [4] J. Richter – *Applied Microsoft .NET Framework Programming*, Microsoft Press, 2002.
- [5] <http://acs.ase.ro/paw>