




Modulul 8

Crearea și distrugerea unui obiect




itacad
you@technology

Microsoft
.NET

Overview

- ▶ Utilizarea constructorilor
- ▶ Inițializarea datelor
- ▶ Obiectele și memoria
- ▶ Administrarea resurselor



2

➤ În acest modul veți învăța ce se întâmplă atunci când este creat un obiect, cum să folosiți constructorii pentru a inițializa obiecte și cum să folosiți destructorii pentru a distruge obiectele. De asemenea, veți afla cum un obiect este distrus și despre modul în care **Garbage Collection** revendică memoria.

➤ În urma acestui capitol studentul va fi capabil să:

- Utilizeze constructorii pentru a inițializa obiectele
- Creeze constructori supraîncărcați cu număr variabil de parametri
- Descrie timpul de viață al unui obiect și ce se întâmplă atunci când este distrus
- Creeze destructori
- Implementeze metoda **Dispose()**

Crearea obiectelor

- ▶ Pasul 1 – Alocarea memoriei
 - ▶ Utilizați cuvântul cheie **new** pentru a alocă memorie pe heap
- ▶ Pasul 2 – Inițializarea obiectului utilizând un constructor
 - ▶ Utilizați numele clasei urmat de paranteze rotunde

```
Date when = new Date();
```

- ▶ Constructorii au rolul de a inițializa câmpurile private
 - ▶ Configurează o stare inițială validă
 - ▶ Asigură comportamentul dorit pentru obiectul creat

3

- Procesul de creare a unui obiect în C# este format din doi pași:
 - Să utilizeze cuvântul cheie **new** pentru a alocă memorie pentru obiect
 - Să scrie un constructor pentru a transforma memoria obținută prin **new** într-un obiect
- Chiar dacă sunt doi pași descriși aceștia sunt realizați într-o singură expresie. De exemplu, dacă **Date** este numele clasei, utilizați următoarea sintaxă:

Date when = new Date();
- **Pasul 1 – alocarea memoriei**
 - Primul pas în crearea unui obiect este alocarea memoriei. Toate obiectele sunt create utilizând operatorul **new**. Nu există excepții la această regulă. Poți să faci asta explicit în cod sau, dacă nu, compilatorul o va face în locul tău.
 - În tabelul următor sunt exemple de cod și ce reprezintă de fapt:

Exemplu de cod	Reprezentare
string s = "Hello";	string s = newstring(new char[]{'H','e','l','l','o'});
int[] array = {1,2,3,4};	int[] array = new int[4]{1,2,3,4};

- **Pasul 2 – inițializarea obiectului utilizând un constructor**
 - Al doilea pas în crearea unui obiect este apelarea constructorului. Constructorul transformă memoria alocată de **new** într-un obiect. Sunt două tipuri de constructori: constructori de instanță și constructori statici. Constructorii de instanță sunt constructorii care inițializează un obiect. Constructorii statici sunt cei care inițializează clasele.

▪ **Cum colaborează operatorul *new* și constructorul de instanță?**

- Este important să înțelegeți această relație. Singurul scop al lui ***new*** este să obțină memorie neinițializată. Singurul scop al constructorului este să inițializeze această memorie și să o tranforme într-un obiect gata de utilizat.
- Deși ***new*** și constructorul au obiective diferite, ca programator nu le poți folosi separat. Acesta este un mod de a asigura faptul că memoria este întotdeauna asociată cu o valoare validă înainte de a fi citită.


itacad
you@technology

Microsoft
.NET

Utilizarea constructorului implicit

► Caracteristici

- Accesibilitate publică
- Aceeași denumire ca a clasei
- Nu are tip, nici măcar *void*
- Nu are argumente
- Inițializează toate câmpurile cu **zero**, **false** sau **null**

► Sintaxă

```
class Date { public Date( ) { ... } }
```

4

► Când creezi un obiect în C#, compilatorul oferă un constructor implicit, dacă nu este creat unul explicit. Să observăm codul următor:

```
class Date
{
    private int ccyy, mm, dd;
}
class Test
{
    static void Main( )
    {
        Date when = new Date( );
        ...
    }
}
```

► Instrucțiunea din interiorul **Test.Main** creează un obiect **Date**, numit **when**, utilizând operatorul *new* și apelând o metodă specială care are același nume ca și clasa. Totuși, clasa **Date** nu are un constructor de instanță declarat (nu are nicio metodă de fapt). Implicit, compilatorul generează un constructor automat.

► Caracteristici ale constructorului implicit

- Conceptual constructorul generat de compilator arată ca în următorul cod:

```
class Date
{
    public Date( )
    {
        ccyy = 0;
    }
}
```

```

        mm = 0;
        dd = 0;
    }
    private int ccyy, mm, dd;
}

```

- Constructorul implicit are următoarele caracteristici:
 - Același nume ca și clasa – prin definiție un constructor este o metodă care are același nume ca și clasa. Aceasta este o definiție naturală și intuitivă și corespunde sintaxei care a fost deja prezentată.
 - Fără tip de returnare – un constructor nu are tip de returnare nici măcar **void**
 - Fără argumente – este posibil să fie declarați constructori ce primesc parametrii. Cu toate acestea, constructorul implicit nu așteaptă niciun argument.
 - Toate câmpurile inițializate cu valoarea nulă corespunzătoare tipului:
 - numerice (**int**, **double**, **decimal**) cu **zero**
 - **bool** cu **false**
 - tipuri referință cu **null**
 - câmpurile dintr-o **struct** cu **zero**
 - Accesibilitate publică – permite ca noi instanțe din acea clasă să fie create

Supraîncărcarea constructorului implicit

- ▶ Constructorul implicit poate să nu fie potrivit.
 - ▶ În această situație nu îl folosiți, scrieți unul nou

```
class Date
{
    public Date( )
    {
        ccyy = 1970;
        mm = 1;
        dd = 1;
    }
    private int ccyy, mm, dd;
}
```

5

➤ Uneori nu este potrivit să folosești constructorul implicit generat automat de compilator. În acest caz este util să creezi propriul constructor care conține doar codul ce inițializează câmpurile cu valori nenule. Orice câmp care nu este inițializat în constructor i se va aplica inițializarea implicită cu zero.

- Există câteva situații în care constructorul implicit nu este corespunzător:
 - Accesul public este, uneori, nedorit – sunt metode care au constructori privați
 - Inițializarea cu zero este, uneori, nedorită
 - Codul invizibil este greu de menținut – nu poți vedea codul constructorului implicit ceea ce se poate transforma, ocazional, într-o problemă, mai ales la depanare

➤ Un exemplu de constructor este următorul:

```
class ImplicitInit
{
    public int a, b;
    public ImplicitInit( )
    {
        a = 42; // b retains implicit initialization to zero
    }
}
class Test
{
    static void Main( )
    {
        ImplicitInit di = new ImplicitInit( );
        Console.WriteLine(di.a); // Writes 42
        Console.WriteLine(di.b); // Writes 0
    }
}
```

Supraîncărcarea unui constructor

- ▶ Constructorii sunt metode și pot fi supraîncărcați
 - ▶ Aceeași clasă, același nume, parametrii diferiți
 - ▶ Permit ca obiectele să fie inițializate în diferite moduri

```
class Date
{
    public Date( ) { ... }
    public Date(int year, int month, int day) { ... }
    ...
}
```

- ▶ **Atenție!**
 - ▶ Dacă scrieți un constructor pentru o clasă, compilatorul nu va mai crea unul implicit

6

➤ Supraîncărcarea este un mod de a declara două sau mai multe metode, cu același nume, în aceeași clasă. Exemplu:

```
class Overload
{
    public void Method( ) { ... }
    public void Method(int x) { ... }
}
class Use
{
    static void Main( )
    {
        Overload o = new Overload( );
        o.Method( );
        o.Method(42);
    }
}
```

➤ În acest cod, două metode cu numele **Method()** sunt declarate în clasa **Overload** și ambele sunt apelate în **Use.Main()**. Nu este nicio ambiguitate pentru că numărul și tipul parametrilor determină despre care dintre metode este vorba.

➤ Inițializarea unui obiect în mai multe feluri a reprezentat o primă motivație în utilizarea supraîncărcării. Constructorii, fiind niște metode mai speciale, se comportă exact ca ele și când vine vorba de supraîncărcare. Aceasta înseamnă că se pot defini mai multe modalități de inițializare a obiectului. Următorul cod este un exemplu:


```
class Overload
{
    public Overload( ) { this.data = -1; }
    public Overload(int x) { this.data = x; }
    private int data;
}
class Use
{
    static void Main( )
    {
        Overload o1 = new Overload( );
        Overload o2 = new Overload(42);
        ...
    }
}
```

➤ Obiectul **o1** este creat utilizând constructorul, care nu primește niciun argument și câmpul privat **data** este inițializat cu -1. Obiectul **o2** este inițializat cu constructorul cu un parametru și **data** va avea valoarea 42.

➤ Vor exista multe cazuri în care nu se dorește inițializarea câmpurilor cu zero. În această situație, puteți să scrieți proprii constructori care primesc parametrii ce sunt utilizați în inițializare. Să considerăm exemplul:

```
class Date
{
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

➤ O problemă cu acest constructor este că va fi foarte ușor să inversezi ordinea parametrilor din neatenție. Exemplu:

Date birthday = new Date(23, 11, 1968); // Error

➤ Acest cod ar trebui să fie: **Date birthday = new Date(23, 11, 1968);** . Compilatorul nu va detecta nicio eroare pentru că toți cei trei parametrii sunt întregi. O metodă de a rezolva această situație este de a utiliza șablonul **Whole Value**. Ați putea să transformați **Year, Month** și **Day** într-un *struct* fiind mai potrivit decât *int* după cum urmează:

```
struct Year
{
    public readonly int value;
    public Year(int value) { this.value = value; }
}
struct Month // Or as an enum
{
    public readonly int value;
    public Month(int value) { this.value = value; }
}
```

```

struct Day
{
    public readonly int value;
    public Day(int value) { this.value = value; }
}
class Date
{
    public Date(Year y, Month m, Day d)
    {
        ccyy = y.value;
        mm = m.value;
        dd = d.value;
    }
    private int ccyy, mm, dd;
}

```

➤ Utilizând *struct* sau *enum* în locul claselor pentru **Day**, **Month**, **Year** se va reduce *overhead*-ul la crearea unui obiect **Date** (va fi explicat mai târziu în acest modul).

➤ Următorul exemplu nu numai că determină erorile de ordine a parametrilor, dar ne și permite să folosim constructori supraîncărcați pentru diferite formate ale datei (U.S., U.K., ISO):

```

class Date
{
    public Date(Year y, Month m, Day d) { ... } // ISO
    public Date(Month m, Day d, Year y) { ... } // US
    public Date(Day d, Month m, Year y) { ... } // UK
    ...
    private int ccyy, mm, dd;
}

```

➤ Dacă declari o clasă cu un constructor, compilatorul nu îl va mai genera pe cel implicit. În exemplul următor, clasa **Date** are un constructor declarat și, de aceea, instrucțiunea **new Date()** nu va compila.

```

class Date
{
    public Date(Year y, Month m, Day d) { ... }
    // No other constructor
    private int ccyy, mm, dd;
}
class Fails
{
    static void Main( )
    {
        Date implicted = new Date( ); // Compile-time error
    }
}

```

➤ Aceasta înseamnă că dacă dorești să creezi obiecte utilizând un constructor implicit, trebuie neapărat ca acel constructor să fie supraîncărcat explicit. Exemplu:

```
class Date
{
    public Date( ) { ... }
    public Date(Year y, Month m, Day d) { ... }
    ...
    private int ccyy, mm, dd;
}
class Succeeds
{
    static void Main( )
    {
        Date implicated = new Date( ); // Okay
    }
}
```


itacad
you@technology

Microsoft
.NET

Liste de inițializare

► Constructorii supraîncărcați pot conține cod duplicat

- Constructorii se vor apela unul pe celălalt
- Utilizați cuvântul cheie **this**

```
class Date
{
    ...
    public Date( ) : this(1970, 1, 1) { }
    public Date(int year, int month, int day) { ... }
}
```

► Pentru eficiență, constructorii vor fi scriși în ordine de la cei complecși la cei simpli

- Cei complecși implementează logica de inițializare, iar cei simpli stabilesc valori standard

7

► Următorul cod este un exemplu de constructori supraîncărcați care conțin cod repetitiv:

```
class Date
{
    public Date( )
    {
        ccyy = 1970;
        mm = 1;
        dd = 1;
    }
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

► Observați repetarea atribuirilor pentru variabilele **ccyy**, **mm** și **dd**. Un mod standard de a evita duplicatele este extragerea codului care se repetă într-o metodă proprie. Exemplu:

```
class Date
{
    public Date( )
    {
        Init(1970, 1, 1);
    }
}
```

```

public Date(int year, int month, int day)
{
    Init(day, month, year);
}
private void Init(int year, int month, int day)
{
    ccyy = year;
    mm = month;
    dd = day;
}
private int ccyy, mm, dd;
}

```

➤ Aceasta este o soluție mai bună decât cea anterioară. Din nefericire, rescrierea constructorilor în felul acesta nu funcționează întotdeauna (spre exemplu dacă ai un câmp *read-only*). Dar C# vine cu o soluție și pentru situația aceasta: **liste de inițializare**.

➤ O listă de inițializare îți permite să scrii un constructor care apelează alt constructor în aceeași clasă. Lista de inițializare se scrie între paranteza rotundă de închidere și acolada de deschidere. Lista de inițializare începe cu ":" și e urmat de cuvântul cheie **this** împreună cu parametrii între paranteze rotunde. Această sintaxă este eficientă și funcționează întotdeauna, nefiind nevoie să mai fie creată o metodă **Init()**.

```

class Date
{
    public Date( ) : this(1970, 1, 1)
    {
    }
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}

```

➤ Există trei restricții de care trebuie să se țină cont atunci când sunt folosite liste de inițializare:

- Listele de inițializare se utilizează doar în constructor:

```

class Point
{
    public Point(int x, int y) { ... }
    // Compile-time error
    public void Init( ) : this(0, 0) { }
}

```

- Nu poate fi scrisă o listă de inițializare care se apelează pe sine.

```

class Point
{
    // Compile-time error
    public Point(int x, int y) : this(x, y) { }
}

```

- Nu poate fi folosit cuvântul cheie **this** ca argument într-o expresie de creare a unui constructor.

```
class Point
{
    // Compile-time error
    public Point( ) : this(X(this), Y(this)) { }
    public Point(int x, int y) { ... }
    private static int X(Point p) { ... }
    private static int Y(Point p) { ... }
}
```

Variabile *read-only* și constante

► Constantă

- Valoarea este obținută la compilare și se setează în momentul declarării variabilei

```
const int maxSize = 100;
```

► Variabilă *read-only*

- Valoarea este obținută la rulare, dar se poate modifica numai în constructori

```
readonly int maxSize = 100;
```

8

- Când folosești constructori trebuie să știi cum să declari variabile *read-only* și constante.

► Utilizarea variabilelor *read-only*

- Un câmp poate fi declarat *read-only* atunci când este definit ca în exemplul următor:

```
readonly int nLoopCount = 10;
```

- Se va obține o eroare de compilare dacă se încearcă schimbarea acestei valori.

► Utilizarea constantelor

- O variabilă constantă reprezintă o valoare constantă care este obținută în timpul compilării. Valoarea unei variabile constante nu poate fi schimbată niciodată.

```
const int speedLimit = 55;
```

- Constantele pot să depindă de alte constante din același program atât timp cât nu este vorba de o dependență circulară. Compilatorul automat evaluează declarațiile constantelor în ordinea corespunzătoare.

Inițializarea câmpurilor *read-only*

- ▶ Câmpurile *read-only* trebuie să fie inițializate
 - ▶ Implicit cu **zero**, **false** sau **null**
 - ▶ Explicit la declararea lor prin inițializarea variabilei
 - ▶ Explicit într-un constructor

```
class CustomArray
{
    public readonly int maxSize;
    public CustomArray(int size)
    { maxSize = size; }
}
```

9

- Există trei metode de inițializare a câmpurilor *read-only*.
 - Inițializarea implicită
 - Inițializarea într-un constructor
 - Inițializarea la declararea câmpului
- **Inițializarea implicită**
 - Constructorii implicați generați de compilator vor inițializa automat toate câmpurile (indiferent dacă sunt *read-only* sau nu) cu valorile lor implicite: **null**, **zero** sau **false**. Exemplu:

```
class SourceFile
{
    public readonly ArrayList lines;
}
class Test
{
    static void Main( )
    {
        SourceFile src = new SourceFile( );
        Console.WriteLine(src.lines == null); // True
    }
}
```

- Nu există niciun constructor **SourceFile**, prin urmare compilatorul va scrie unul pentru tine care va inițializa **line** cu **null**.
- Dacă declari propriul constructor și nu inițializezi explicit câmpul *read-only* compilatorul tot îl va inițializa automat. Exemplu:

```
class SourceFile
{
    public SourceFile( ) { }
    public readonly ArrayList lines;
}
```



```
class Test
{
    static void Main( )
    {
        SourceFile src = new SourceFile( );
        Console.WriteLine(src.lines == null); // Still true
    }
}
```

- În acest caz, câmpul *read-only* este inițializat cu *null* și va rămâne *null* pentru că nu poți să reasignezi acest câmp.

➤ Inițializarea într-un constructor

- Poți explicit să atribui o valoare unui câmp *read-only* în corpul unui constructor. Exemplu:

```
class SourceFile
{
    public SourceFile( )
    {
        lines = new ArrayList( );
    }
    private readonly ArrayList lines;
}
```

- Instrucțiunea din corpul constructorului arată exact ca o asignare a variabilei *lines*, ceea ce în mod normal nu este permis pentru că variabila e *read-only*. Totuși, instrucțiunea compilează deoarece compilatorul observă că aceasta se execută în interiorul unui constructor așa că o tratează ca pe o inițializare.
- Un avantaj al inițializării în acest fel este că poți folosi parametrul constructorului cu operatorul **new**. Exemplu:

```
class SourceFile
{
    public SourceFile(int suggestedSize)
    {
        lines = new ArrayList(suggestedSize);
    }
    private readonly ArrayList lines;
}
```

➤ Inițializarea explicită la declararea câmpului

- Un câmp poate fi inițializat direct la declararea sa.

```
class SourceFile
{
    public SourceFile( ){
        private readonly ArrayList lines = new ArrayList( );
    }
}
```

- Echivalentul pentru compilator al codului de mai sus este următorul:

```
class SourceFile
{
    public SourceFile( )
    {
        lines = new ArrayList( );
    }
    private readonly ArrayList lines;
}
```

Constructorul pentru Struct

- | | |
|--|---|
| <ul style="list-style-type: none"> ▶ Compilerul <ul style="list-style-type: none"> ▶ Întotdeauna generează un constructor implicit ▶ Automat se inițializează toate câmpurile cu zero | <ul style="list-style-type: none"> ▶ Programatorul <ul style="list-style-type: none"> ▶ Poate declara constructori cu parametrii ▶ Nu sunt inițializate implicit câmpurile cu zero ▶ Nu poate declara un constructor implicit ▶ Nu poate declara un constructor <i>protected</i> |
|--|---|

10

- Sintaxa pentru constructorul unei structuri este la fel ca la clasă. Exemplu:

```
struct Point
{
    public Point(int x, int y) { ... }
}
```

- Deși sintaxa pentru constructorul unei clase și **struct** este aceeași, apar anumite restricții pentru structură:

- Compilerul creează întotdeauna un constructor implicit pentru **struct** indiferent dacă scrii unul propriu sau nu (aceasta este diferit față de clase unde compilerul nu mai crea un constructor implicit dacă exista deja unul declarat explicit). Constructorul implicit generat de compiler inițializează toate câmpurile cu următoarele valori: **zero**, **null** sau **false**.

```
struct SPoint
{
    public SPoint(int x, int y) { ... }
    static void Main( )
    {
        // Okay
        SPoint p = new SPoint( );
    }
}

class CPoint
{
    public CPoint(int x, int y) { ... }
    static void Main( )
    {
        // Compile-time error
        CPoint p = new CPoint( );
    }
}
```

- Aceasta înseamnă că instrucțiunea ***SPoint p = new SPoint();*** crează o valoare ***struct*** pe stivă și inițializează toate câmpurile cu ***zero***.
- Totuși declarația ***SPoint p;*** crează o valoare ***struct*** pe stivă dar nu inițializează câmpurile. Exemplu:

```
struct SPoint
{
    public int x, y;
    static void Main( )
    {
        SPoint p1;
        Console.WriteLine(p1.x); // Compile-time error
        SPoint p2;
        p2.x = 0;
        Console.WriteLine(p2.x); // Okay
    }
}
```

- Nu poți declara un constructor implicit într-un ***struct***.

- Motivul pentru această restricție apare deoarece compilatorul va crea întotdeauna un constructor implicit pentru ***struct*** ceea ce va duce la o definiție duplicată.

```
class CPoint
{
    // Okay because CPoint is a class
    public CPoint( ) { ... }
}
struct SPoint
{
    // Compile-time error because SPoint is a struct
    public SPoint( ) { ... }
}
```

- Poți declara încă un constructor atât timp cât diferă prin lista de argumente de cel implicit.

```
struct SPoint
{
    public SPoint(int x, int y) { ... }
}
```

- Nu poți declara un constructor ***protected*** într-un ***struct***.

- Motivul pentru această restricție este că nu poți să derivezi alte clase sau alte structuri dintr-o ***struct***, de aceea accesul protejat nu are sens.

```
class CPoint
{
    // Okay
    protected CPoint(int x, int y) { ... }
}
struct SPoint
{
    // Compile-time error
    protected SPoint(int x, int y) { ... }
}
```

- Trebuie ca toate câmpurile unei structuri să fie inițializate.
 - Dacă declari un constructor de clasă propriu care eșuează în a inițializa un câmp, compilatorul se va asigura ca acel câmp va beneficia de inițializarea implicită cu zero. Exemplu:

```
class CPoint
{
    private int x, y;
    public CPoint(int x, int y) { /*nothing*/ }
    // Okay. Compiler ensures that x and y are initialized to
    // zero.
}
```

- Spre deosebire de o clasă, dacă un constructor de **struct** eșuează în a inițializa un câmp, compilatorul va genera o eroare de compilare. Exemplu:

```
struct SPoint1 // Okay: initialized when declared
{
    private int x,y;
    public SPoint1(int a, int b) { }
}
struct SPoint2 // Okay: initialized in constructor
{
    private int x, y;
    public SPoint2(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```



Construcții private

- ▶ Limitează construcția obiectelor din afara clasei
 - ▶ Pot fi apelate metode ale obiectului
 - ▶ Pot fi apelate metode statice
 - ▶ Un mod util de a implementa clase cu funcții procedurale

```
public class Math
{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
    private Math( ) { }
}
```

11

➤ Programarea orientată pe obiecte oferă o paradigmă puternică pentru structurarea software în foarte multe domenii. Totuși, nu este o paradigmă aplicabilă universal. De exemplu, nu este nimic legat de orientarea pe obiecte în calculul sinusului și al cosinusului al unui număr zecimal.

➤ Declararea funcțiilor

- Cel mai intuitiv mod de a calcula sin sau cos este să folosești funcții globale declarate în afara obiectului, precum:

```
double Cos(double x) { ... }
```

```
double Sin(double x) { ... }
```

- Codul anterior nu este permis în C#. Funcțiile globale sunt permise în limbaje procedurale, precum C/C++, dar nu și în C#, unde trebuie declarate în interiorul unei clase sau al unei structuri.

```
class Math
```

```
{
```

```
    public double Cos(double x) { ... }
```

```
    public double Sin(double x) { ... }
```

```
}
```

➤ Declararea metodelor statice

- Problema cu cele două funcții în codul anterior este că pot fi folosite numai creând un obiect al clasei **Math**:

```
class Cumbersome
```

```
{
```

```
    static void Main( )
```

```
{
```

```
        Math m = new Math( );
```

```
        double answer;
```

```
        answer = m.Cos(42.0);
```

```
    }
```

```
}
```

- Cu toate acestea, problema poate fi rezolvată declarând cele două funcții statice în clasa **Math** ca în codul următor:

```
class Math
{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
    private Math( ) { ... }
}
class LessCumbersome
{
    static void Main( )
    {
        double answer = Math.Cos(42.0);
    }
}
```

➤ Beneficiile metodelor statice

- Dacă declarați **Cos** ca metodă statică, sintaxa pentru utilizare devine:
 - Mai simplă: **Math.Cos()**;
 - Mai rapidă: nu este nevoie să mai creezi un obiect pentru a accesa metoda.
- O mică problemă mai rămâne totuși. Compilatorul va genera un constructor implicit cu accesibilitate publică, permițându-ți să creezi obiecte **Math**. Aceste obiecte nu au niciun folos, întrucât clasa conține numai metode statice. Există două posibilități prin care se poate rezolva această situație:
 - Clasa **Math** să fie declarată ca abstractă. Nu este o idee bună, deoarece scopul unei clase abstracte este să fie derivată.
 - Clasa **Math** să aibă un constructor privat. Aceasta este soluția corectă. Declarând un constructor pentru clasă, generatorul nu va mai genera unul implicit și pentru că este privat clasa nu va putea fi instanțiată. Din cauză că **Math** conține un constructor privat, clasa nu va putea să fie de bază pentru alte clase derivate.

Constructori statici

► Scop

- Apelați la compilare atunci când se încarcă respectiva clasă
- Utilizați pentru a inițializa câmpuri statice
- Sunt apelați garantat înainte de constructorii de obiect

► Restricții

- Nu pot fi apelați de codul obiect (cod non-static)
- Nu pot avea un grad de accesibilitate (*private* sau *public*)
- Nu au parametri

12

► C# este un limbaj dinamic. Când **CLR** rulează un program **.NET**, întâlnește uneori cod care utilizează o clasă ce nu a fost încă încărcată. În aceste situații, execuția este suspendată, clasa este încărcată dinamic și apoi execuția continuă.

► C# se poate asigura mereu că o clasă este inițializată înainte de a fi folosită în cod. Acest lucru este realizat prin constructorii statici.

► Declararea constructorilor statici este la fel ca la cei obișnuiți numai că apare cuvântul cheie **static** în definiție:

```
class Example
{
    static Example() { ... }
}
```

► După ce o clasă este încărcată pentru a fi folosită, se execută constructorii statici pentru acea clasă. Datorită acestui proces ai garantat întotdeauna faptul că o clasă va fi inițializată înainte de utilizare. Sincronizarea exactă a executării constructorilor statici depinde de implementare, dar se ghidează după următoarele reguli:

- Constructorul static este executat înainte ca o instanță a acelei clase să fie create.
- Constructorul static este executat înainte ca orice membru static al clasei să fie referit.
- Constructorul static este executat cel mult o singură dată în timpul instanțierii unui program.

► Cea mai comună utilizare a unui constructor static este inițializarea câmpurilor statice ale clasei.

```
class Example
{
    private static Wibble w = new Wibble();
}
```

- Codul de mai sus este convertit de compilator în:

```
class Example
{
    static Example( )
    {
        w = new Wibble( );
    }
    private static Wibble w;
}
```

- Restricții privind constructorii statici:

- Nu poți apela un constructor static.
 - Un constructor static trebuie apelat înainte ca orice instanță a clasei să fie referită în cod. Dacă această responsabilitate ar fi fost dată programatorului, ci nu compilatorului, cel mai probabil ar fi apărut erori: ar fi uitat să apeleze constructorul sau l-ar fi apelat mai mult decât o singură dată. Pentru a evita acestea, numai **.NET runtime** apelează constructorii statici.

```
class Point
{
    static Point( ) { ... }
    static void Main( )
    {
        Point.Point( ); // Compile-time error
    }
}
```

- Nu poți declara un constructor static cu un modificador de accesibilitate.
 - Deoarece nu poți apela constructorul, declararea acestuia cu un modificador de accesibilitate nu are sens și cauzează o eroare de compilare.

```
class Point
{
    public static Point( ) { ... } // Compile-time error
}
```

- Nu poți declara un constructor static cu parametri.
 - Deoarece nu poți apela constructorul, declararea acestuia cu parametri nu are sens și cauzează o eroare de compilare. De asemenea, este implicat faptul că nu poți avea constructori statici supraîncărcați.

```
class Point
{
    static Point(int x) { ... } // Compile-time error
}
```

- Nu poți folosi **this** într-un constructor static.
 - Deoarece constructorul static inițializează clasa, ci nu obiectele clasei, nu are o referință implicită la **this**.

```
class Point
{
    private int x, y;
    static Point( ) : this(0,0) // Compile-time error
    {
        this.x = 0; // Compile-time error
        this.y = 0; // Compile-time error
    }
}
```


Timpul de viață al unui obiect

- ▶ Crearea unui obiect
 - ▶ Se alocă memorie utilizând **new**
 - ▶ Se inițializează acea memorie utilizând constructorul
- ▶ Utilizarea unui obiect
 - ▶ Se pot apela metode
- ▶ Ștergerea unui obiect
 - ▶ Obiectul nu va mai putea fi folosit
 - ▶ Memoria este dealocată

13

➤ Crearea obiectelor

- În prima secțiune a lecției ați învățat cum să creați un obiect. Cei doi pași sunt:
 - Utilizarea lui **new** pentru a alocă memorie.
 - Apelarea unui constructor pentru a transforma memoria obținută într-un obiect.

➤ Distrugerea obiectelor

- Distrugerea unui obiect este tot un proces în doi pași:
 - De-inițializarea unui obiect: se transformă obiectul în memorie utilizând un destructor (inversul unui constructor). Poți controla ce se întâmplă la distrugerea obiectului scriind propriul cod pentru destructor și pentru metoda **finalize()**.
 - Memoria dealocată, este dată înapoi **heap**-ului (inversul procesului realizat de **new**).


 itacad
 you@technology

 Microsoft
 .NET

Obiectul și sfera de vizibilitate

- ▶ O variabilă locală este vizibilă în domeniul în care este alocată
 - ▶ Timp de viață scurt
 - ▶ Determinism – la creare și distrugere
- ▶ Un obiect dinamic nu depinde de domeniul său
 - ▶ Un timp de viață mai lung
 - ▶ Nedeterminism – la distrugere

14

➤ Spre deosebire de valori precum **int** sau **struct**, care sunt create pe stivă și distruse la sfârșitul sferei de vizibilitate a acestora, obiectele sunt create pe **heap** și nu sunt distruse neapărat cum se întâmplă la valori.

➤ Valori

- Timpul de viață al unei valori este strâns legat de codul în care este declarată și vizibilă. De exemplu, în codul următor, trei valori sunt în interiorul instrucțiunii **for** și sunt utilizate în afara acesteia. Evident vor rezulta în erori de compilare.

```

struct Point { public int x, y; }
enum Season { Spring, Summer, Fall, Winter }
class Example
{
    void Method(int limit)
    {
        for (int i = 0; i < limit; i++) {
            int x = 42;
            Point p = new Point( );
            Season s = Season.Winter;
            ...
        }
        x = 42; // Compile-time error
        p = new Point( ); // Compile-time error
        s = Season.Winter; // Compile-time error
    }
}
  
```

- Aceasta înseamnă că valorile au următoarele caracteristici:
 - Creare și distrugere deterministice. O variabilă este creată la declarare și distrusă la sfârșitul codului în care are vizibilitate. Acești timpi sunt cunoscuți, determinați.

- De obicei, timp de viață foarte scurt: în general cam cât durează execuția unei metode, sau cel mult al unui program.

➤ Obiecte

- Timpul de viață al unui obiect nu este legat de sfera lui de vizibilitate. Obiectele sunt inițializate în memoria **heap**. De exemplu, în codul următor, obiectul **eg** este creat în interiorul instrucțiunii **for**, dar asta nu înseamnă că este distrus la sfârșitul instrucțiunii:

```
class Example
{
    void Method(int limit)
    {
        for (int i = 0; i < limit; i++) {
            Example eg = new Example( );
            ...
        }
        // eg is out of scope
        // Does eg still exist?
        // Does the object still exist?
    }
}
```

- Obiectele au următoarele caracteristici:
 - Distrugere nedeterministică. Spre deosebire de valori, un obiect nu este distrus după ce iese din sfera de vizibilitate. Crearea unui obiect este deterministică, dar distrugerea nu poate fi determinată.
 - Timp de viață mai lung


itacad
you@technology

Microsoft
.NET

Garbage Collection

- ▶ Obiectele nu pot fi distruse explicit
 - ▶ În C# nu există opusul lui **new** pentru că reprezintă o sursă de erori, așa cum se întâmplă în limbajul C++
- ▶ GB distruge obiectele pentru tine
 - ▶ Identifică obiectele care nu mai sunt referențiate
 - ▶ Se recuperează memoria *heap* asociată obiectului din zona
 - ▶ În general, acest lucru se întâmplă atunci când se ajunge la memorie puțină

15

➤ În foarte multe limbaje de programare poți controla explicit când un obiect este distrus. De exemplu în C++ poți folosi **delete** pentru a finaliza acel obiect. În C# nu poți face explicit acest lucru. Această restricție este în foarte multe feluri utilă, deoarece programatorii folosesc uneori greșit abilitatea de a distruge un obiect explicit.

➤ Cele mai comune greșeli sunt:

- Uiti să distrugi obiectele
 - De cele mai multe ori, programatorii uită să scrie codul pentru distrugerea obiectului. Această situație apare în C++ și duce la rularea mult mai înceată a programului și folosirea a foarte multă memorie.
- Încerci să distrugi un obiect mai mult decât o dată.
 - Uneori, accidental, programatorii încearcă să ștergă obiectul de mai multe ori ceea ce duce la un **bug** serios. Problema este că atunci când distrugi un obiect memoria este revendicată și cu ajutorul lui **new** poate fi transformată în alt obiect, probabil aparținând unei clase complet diferite. Când încerci să distrugi obiectul a doua oară acea memorie se referă la cu totul alt obiect.
- Distrugerea unui obiect activ
 - Se încearcă distrugerea unui obiect care încă mai este referit în altă parte a programului ducând la crearea unei erori foarte grave.

➤ În C# nu poți distruge obiectele explicit. În schimb, în C# există **Garbage Collection** care o va face automatic pentru tine.

➤ **Garbage Collection** asigură că:

- Obiectele sunt distruse
 - Totuși, **GB** nu specifică exact când va face asta.
- Obiectele sunt distruse doar o dată
- Numai obiectele nefolosite sunt distruse
 - **GB** se asigură că un obiect nu este distrus în situația în care alt obiect păstrează o referință către el. Aceasta este o operație foarte consumatoare, de aceea, numai dacă este nevoie (se ajunge la puțină memorie), **GB** șterge obiecte.


itacad
you@technology

Microsoft
.NET

Distrugerea unui obiect

- ▶ Acțiunile finale ale fiecărui obiect sunt diferite
 - ▶ Nu pot fi determinate de GB
- ▶ Obiectele în .NET au o metodă **Finalize()**
 - ▶ Utilizată pentru eliberarea resurselor utilizate de obiect
 - ▶ Nu se poate apela sau supraîncărca metoda **Object.Finalize()**
 - ▶ Pentru modificarea acesteia se va scrie un destructor
 - ▶ GB va apela destructorul înainte de a revendica memoria

16

- Deja ați văzut că distrugerea unui obiect este un proces în doi pași. În primul pas obiectul este convertit din nou în memorie. În al doilea pas, memoria este returnată **heap**-ului pentru a fi reciclată. **GB** automatizează al doilea pas din proces pentru tine.
- Acțiunea de a transforma un obiect în memorie brută depinde doar de obiect. **GB** nu automatizează și primul pas. Dacă sunt anumite instrucțiuni pe care vrei ca obiectul să le execute când este preluat de **GB**, exact înainte ca memoria să fie revendicată, trebuie să scrii un **destructor**.
- Când **GB** distruge un anumit obiect, verifică mai întâi dacă acea clasă are un **destructor** sau o metodă **finalize()**. Dacă da, le va apela pe acestea mai întâi, după care va transforma obiectul în memorie brută.

Scrierea unui destructor

- ▶ Curățarea resurselor declarate în constructor
- ▶ Are sintaxă proprie:
 - ▶ Fără modificatori de accesibilitate
 - ▶ Fără tip (inclusiv fără *void*)
 - ▶ Același nume ca respectiva clasă precedat de ~
 - ▶ Fără parametri

```
class SourceFile
{
    ~SourceFile( ) { ... }
}
```

17

- Pentru a șterge un obiect, poți să scrii un destructor. În C#, metoda **Finalize()** nu este direct accesibilă și nu poate fi apelată sau supraîncărcată.
- Exemplul următor este un destructor pentru o clasă numită **SourceFile**

```
~ SourceFile( ) {
    // Perform cleanup
}
```
- Un destructor nu poate avea:
 - un determinant de accesibilitate
 - Nu programatorul apelează destructorul, ci **Garbage Collection**.
 - un tip de returnare
 - parametri

Sincronizarea destructorului

- ▶ Ordinea și sincronizarea destructorului este nedefinită
 - ▶ Nu în mod necesar inversul constructorului
- ▶ Destructorii sunt apelați garantat
 - ▶ Nu vă bazați pe sincronizare
- ▶ Evitați destructorii dacă este posibil
 - ▶ Costuri de performanță
 - ▶ Complexitate
 - ▶ Întârzierea eliberării memoriei

➤ Am observat că **Garbage Collection** este responsabil de distrugerea obiectelor atunci când ele nu mai sunt folosite deloc. Aceasta este complet diferit față de alte limbaje de programare precum C++, în care programatorul era direct responsabil de distrugerea obiectelor. Transferul responsabilității de la programator este un lucru bun, însă nu poți preciza cu exactitate când va avea loc. Termenul folosit pentru această situație este **“finalizare non-deterministică”**.

➤ În C#, ordinea în care obiectele sunt create nu determină ordinea în care acestea sunt distruse. Totuși, în practică, nu este o problemă pentru că **GB** asigură faptul că un obiect nu este distrus dacă există referințe către el. Spre exemplu, dacă un obiect are o referință către un al doilea obiect atunci cel de-al doilea obiect nu va fi niciodată distrus înaintea primului.

➤ Evitați destructorii pe cât posibil. Finalizarea adaugă *overhead*, complexitate și întârzie revendicarea memoriei. Implementați destructori sau o metodă **finalize()** doar dacă este absolut necesar.



Interfața **IDisposable** și metoda **Dispose()**

- ▶ Pentru a revendica o resursă:
 - ▶ Se implementează interfața **IDisposable** care dispune de metoda **Dispose()**
 - ▶ Apelați metoda **GC.SuppressFinalize** pentru a scoate din calcul monitorizarea referinței de către GC
 - ▶ Asigurați-vă că nu încercați să folosiți o resursă deja revendicată

19

➤ Memoria este cea mai folosită resursă de către programe și te poți baza pe **GB** să o revendice pentru tine atunci când **heap**-ul are puține elemente. Oricum, memoria nu este singura resursă existând și altele: **files handler**, **mutex locks**. De multe ori aceste resurse sunt mult mai limitate decât memoria și au nevoie să fie eliberate repede.

➤ În aceste situații nu te poți baza pe **GB** să execute eliberarea memoriei deoarece timpul la care face acest lucru nu este determinat. În schimb, poți scrie o metodă publică **Dispose()** care eliberează resursa, dar trebuie să fii atent să o apelezi în cod la momentul potrivit.

➤ Pentru a distruge un obiect utilizând metoda **Dispose()** trebuie:

- să fie moștenită interfața **IDisposable**
- să fie implementată metoda **Dispose()**
- să vă asigurați că metoda **Dispose()** poate fi apelată de mai multe ori
- să apelați **SuppressFinalize()**

➤ Exemplu:

```
using System;
using System.ComponentModel;
```

```
// The following example demonstrates how to create
// a resource class that implements the IDisposable interface
// and the IDisposable.Dispose method.
```

```
public class DisposeExample
{
```

```
// A base class that implements IDisposable.
// By implementing IDisposable, you are announcing that
// instances of this type allocate scarce resources.
public class MyResource : IDisposable
{
    // Pointer to an external unmanaged resource.
    private IntPtr handle;
    // Other managed resource this class uses.
    private Component component = new Component();
    // Track whether Dispose has been called.
    private bool disposed = false;

    // The class constructor.
    public MyResource(IntPtr handle)
    {
        this.handle = handle;
    }

    // Implement IDisposable.
    // Do not make this method virtual.
    // A derived class should not be able to override this method.
    public void Dispose()
    {
        Dispose(true);
        // This object will be cleaned up by the Dispose method.
        // Therefore, you should call GC.SuppressFinalize to
        // take this object off the finalization queue
        // and prevent finalization code for this object
        // from executing a second time.
        GC.SuppressFinalize(this);
    }

    // Dispose(bool disposing) executes in two distinct scenarios.
    // If disposing equals true, the method has been called directly
    // or indirectly by a user's code. Managed and unmanaged resources
    // can be disposed.
    // If disposing equals false, the method has been called by the
    // runtime from inside the finalizer and you should not reference
    // other objects. Only unmanaged resources can be disposed.
    protected virtual void Dispose(bool disposing)
    {
        // Check to see if Dispose has already been called.
        if (!this.disposed)
        {
            // If disposing equals true, dispose all managed
            // and unmanaged resources.
            if (disposing)
            {
                // Dispose managed resources.
                component.Dispose();
            }
        }
    }
}
```

```
// Call the appropriate methods to clean up
// unmanaged resources here.
// If disposing is false,
// only the following code is executed.
CloseHandle(handle);
handle = IntPtr.Zero;

// Note disposing has been done.
disposed = true;

    }
}

// Use interop to call the method necessary
// to clean up the unmanaged resource.
[System.Runtime.InteropServices.DllImport("Kernel32")]
private extern static Boolean CloseHandle(IntPtr handle);

// Use C# destructor syntax for finalization code.
// This destructor will run only if the Dispose method
// does not get called.
// It gives your base class the opportunity to finalize.
// Do not provide destructors in types derived from this class.
~MyResource()
{
    // Do not re-create Dispose clean-up code here.
    // Calling Dispose(false) is optimal in terms of
    // readability and maintainability.
    Dispose(false);
}

}

public static void Main()
{
    // Insert code here to create
    // and use the MyResource object.
}
}
```


itacad
you@technology

Microsoft
.NET

Instrucțiunea **using**

- ▶ Este folosită pentru a dezaloca instantaneu obiectul din memoria *heap*

- ▶ Sintaxă

```
using (Resource r1 = new Resource( ))
{
    r1.Method( );
}
```

- ▶ **Dispose()** este automat apelată la sfârșitul unui bloc *using*
- ▶ Numai obiectele care implementează **IDisposable** pot fi folosite împreună cu comanda *using*

20

- Instrucțiunea **using** definește un domeniu de cod la sfârșitul căruia obiectul va fi terminat.

- Poți crea o instanță în interiorul instrucțiunii **using** pentru a te asigura că metoda **Dispose()** este apelată imediat ce **using** a terminat de rulat.

- În exemplul următor, **Resource** este un tip referință ce implementează interfața **IDisposable**:

```
using (Resource r1 = new Resource( )) {
    r1.Test( );
}
```

- Codul de mai sus este semantic echivalent cu:

```
Resource r1 = new Resource( );
try {
    r1.Test( );
}
finally {
    if (r1 != null) ((IDisposable)r1).Dispose( );
}
```



itacademy
you@technology

Sumar

- ▶ Utilizarea constructorilor
- ▶ Inițializarea datelor
- ▶ Obiectele și memoria
- ▶ Administrarea resurselor



21

➤ **Cerințe:**

- Declarați o clasă numită **Date** cu un constructor public care așteaptă trei parametri de tip **int** numiți **year**, **month**, **day**.
- Va genera compilatorul un constructor implicit pentru clasa **Date** pe care ai declarat-o mai sus? Dar dacă **Date** ar fi fost o structură cu aceleași trei câmpuri?
- Ce metodă apelează **GB** înainte de a recicla memoria obiectului înapoi în **heap**?
- Care este scopul instrucțiunii **using**?