### Modulul 13. Gestiunea memoriei și garbage collection în C#.

#### Ciclul de viață al unui obiect

Ciclul de viață de viață al unui obiect este intervalul de timp dintre momentul când obiectul a fost creat(și a fost alocată memorie pentru el) și momentul când acesta a fost distrus(și memoria a fost dealocată).



**Crearea obiectelor:**

În C# ciclul de viață al unui obiect începe când folosim cuvântul cheie **new,** ceea ce în esență denotă un constructor. În esență toate clasele în C# reprezintă doar containere de logică, aceasta nu va fi executată până când nu va fi creat un obiect. Când obiectul este creat și memoria este alocată acesta începe să execute toate comenzile care le primește de la clasă.

**Observație: Teoretic putem crea un număr nelimitat de obiecte care aparțin aceleiași clase. Practic suntem limitați de resursele dispozitivului.**

**Distrugerea obiectelor:**

Spre deosebire de C++ care vă permite să apelați destructorii manual pentru a distruge obiectele, în C# acest lucru se execută automat de GC. Ceea ce automatizează gestiunea memoriei spre deosebire de unele limbaje de nivel mai jos. Acest lucru face C# mult mai eficient pentru o mare majoritate de programatori care nu doresc să gestioneze memoria manual.

#### Ce este garbage collection?

Garbage collection este un sistem care permite managementul automat al memoriei în C#(și alte limbaje de programare) acesta nu permite în mod direct programatorilor să folosească destructorul. Motivele pentru implementarea acestui mecanism sunt următoarele scenarii:

* Un programator poate totdeauna să uite să distrugă un obiect.
* Alocarea memoriei manuală este un proces îndelungat și anevoios(mai ales pentru programele mari)
* Dacă un obiect a fost distrus accidental și încercăm să îl accesăm primim excepție.
* Cazul în care încercăm să folosim un destructor în timp ce obiectul încă execută un careva proces.
* Dacă încercăm să distrugem un obiect care a fost distrus deja.

Am expus mai sus motivele potențiale pentru care garbage collection este utilizat, deci cum funcționează acesta și cum previne aceste scenarii? **(GC = Garbage collector)**

* Garbage collector verifică dacă obiectul execută un proces, dacă da acesta nu va fi distrus
* GC verifică dacă obiectul a fost distrus înainte de a folosi destructorul(pentru a nu încerca să distrugă un obiect inexistent)
* GC asigură că toate obiectele la care se face referință în alte obiecte nu vor fi distruse.
* GC include unele metode sigure pentru programatori de încerca să elibereze memoria.

Este necesar de menționat că Garbage Collection este un sistem implementat în platforma .NET, nu în C#. Din acest motiv orice limbaj folosit în .NET (C++, Visual Basic etc.) va avea acces automat la acest mecanism.

#### Clasa System.GC

Clasa System.GC reprezintă Garbage collection și include mai multe metode utile pentru a permite programatorilor să controleze procesul de alocare a memoriei într-un mod sigur.

Metode din clasa GC:

1. **GC.Collect()** - trimite un request către garbage collector pentru a elibera memoria de la toate obiectele care nu mai execută procese și nu au referințe active **(Se folosește pentru a forța GC de a colecta obiectele inactive pentru că în mod normal GC colectează doar când memoria este încărcată)**.
2. **GC.KeepAlive(object obj)** - indică către GC că un obiect nu trebuie distrus, acest lucru poate fi utilizat când vom avea nevoie să reutilizăm un anumit obiect.
3. **GC.ReRegisterForFinalize(object obj)** - indică că un obiect trebuie adăugat în lista de distrugere(manual). **Nu se recomandă utilizarea acestei metode pentru că GC gestionează bine starea obiectelor fără implicarea programatorilor(**Unicul scenariu în care acest lucru este acceptabil este atunci când am folosit KeepAlive pentru obiectul dat preventiv.
4. **GC.SuppressFinalize(object obj)** - indică către GC că obiectul dat nu va folosi destructorul, chiar dacă memoria depășește limita stabilită**(vom folosi metoda atunci când vrem să adăugăm un destructor manual).**
5. **GC.WaitForPendingFinalizers()** - blochează aplicația până când toate obiectele inactive sunt distruse.

#### Destructor și metoda Finalize

Precum am menționat anterior, nu putem invoca manual destructorii(**Finalizers**) în C#, aceștia sunt executați de GC. Dar putem defini o careva logică care se va executa când obiectul va fi distrus.

**Sintaxa:**

|  |
| --- |
| class Exemplu  {  // Destructor  ~Exemplu()  {  Console.WriteLine(“Destructor Invocat”);  }  } |

Precum este demonstrat mai sus, pentru a defini o logică adițională pentru un constructor vom folosi semnul **~** urmat de numele clasei.

**Puncte importante despre destructori:**

* Destructorii nu pot fi definiți în structuri, doar clase.
* O clasă poate avea doar un singur destructor.
* Destructorii nu pot fi moșteniți sau rescriși(overloaded).
* Destructorii nu pot fi apelați manual, doar automat de GC.
* Destructorii nu pot primi parametri.

Deși GC “știe” când să distrugă obiectele care nu mai sunt necesare, acesta nu cunoaște logica programului. De exemplu când un obiect a atașat o metodă către un eveniment din alt obiect, trebuie să ne asigurăm că acest eveniment va fi eliminat, ceea ce putem face cu ajutorul destructorului.

Problema cu destructoarele în C# este că acestea sunt apelate în mod nedeterministic, ceea ce semnifică că GC poate decide că obiectul dat nu trebuie necesat distrus după ce și-a executat funcțiile doar din cauza că cantitatea de memorie liberă este suficientă.

**Exemplu:**

|  |
| --- |
| namespace GestiuneaMemoriei  {  class Exemplu  {  public Exemplu() { WriteLine("Constructor");}  // Destructor  ~Exemplu() { WriteLine("Destructor");}  }  class Program  {  static void Main()  {  Details();  GC.Collect();  }  public static void Details()  {  // Created instance of the class  Exemplu ex = new Exemplu();  }  }  } |

Exemplu 2:

|  |
| --- |
| public class ExampleClass  {  Stopwatch sw;  public ExampleClass()  {  sw = Stopwatch.StartNew();  WriteLine("Instantiated object");  }  public void ShowDuration()  {  WriteLine("This instance of {0} has been in existence for {1}", this, sw.Elapsed);  }  ~ExampleClass()  {  WriteLine("Finalizing object");  sw.Stop();  WriteLine("This instance of {0} has been in existence for {1}", this, sw.Elapsed);  }  }  public class Demo  {  public static void Main()  {  Details();  GC.Collect();  }  public static void Details()  {  ExampleClass ex = new ExampleClass();  ex.ShowDuration();  }  } |

După cum vedeți în exemplele de mai sus putem folosi destructorul apelat pentru a printa mesajul Destructor la consolă. Dar pentru a forța destructor să funcționeze am fost nevoiți să utilizăm metoda GC.Collect, și să folosim constructorul în interiorul unei metode, altfel destructorul nu va fi invocat.

Probleme pentru rezolvare.

### **Program 1:**

1. Pe baza exemplului de mai sus creați un program cu 3 clase externe.
2. Pentru fiecare clasă creați un destructor individual cu un mesaj specific.
3. În clasa care va executa metoda Main, creați câte un obiect din fiecare clasă, apoi folosiți GC.Collect() pentru a elibera memoria și invoca destructorii.

### **Opțional:**

1. Afișați timpul de viață al fiecărui obiect.

### **Program 2:**

1. Creați un program în care veți avea 2 clase.
2. Prima clasă va conține metoda Main și va avea un eveniment.
3. A doua clasă va avea un constructor, un destructor și o metodă care va afișa ceva la consolă:
4. În constructor vom atașa metoda din clasă la evenimentul din clasa 1.
5. În destructor vom scoate metoda din evenimentul din clasa 1.
6. În clasa 1, metoda Main, creați un obiect din clasa 2.
7. Invocați evenimentul.
8. Folosiți GC.Collect(); pentru a elibera memoria.
9. Invocați evenimentul din nou.

#### Metoda Dispose și interfața IDisposable

Deși destructorii permit un anumit nivel de control asupra gestiunii memoriei, aceștia nu pot fi apelați anume atunci când avem nevoie, garbage collector decide automat când este nevoie de a elibera un obiect.

Aceste neajunsuri pot fi eliminate cu ajutorul metodei Dispose.

Metoda Dispose face parte din interfața IDisposable și reprezintă o modalitate sigură de a elibera memoria alocată unui obiect.

Pentru a folosi metoda Dispose o clasă trebuie să moștenească de la interfața IDisposable.

Spre deosebire de destructor, care este o metodă implicită(automată) de a elibera resurse, Dispose() permite programatorilor un nivel mai mare de control asupra execuției.

**Metoda Dispose este folosită în următoarele cazuri:**

* Când știm sigur că un obiect și-a executat rolul și nu va mai fi utilizat.
* Când obiectul nostru conține o legătură cu o sursă externă(de exemplu BD, fișier etc.) de care nu mai avem nevoie.

Pentru a implementa metoda Dispose corect se recomandă următoarea sintaxă:

|  |
| --- |
| namespace Dispose  {  class FinalizeDemo : IDisposable  {  private bool disposed;  //Constructor si destructor  public FinalizeDemo() { WriteLine("Constructor"); }  ~FinalizeDemo() { WriteLine("Destructor"); Dispose(False);}  //Implementarea metodei Dispose  public void Dispose()  {  Dispose(true);  GC.SuppressFinalize(this); //Blocam executia destructorului pentru ca am utilizat metoda Dispose  disposed = true;  }  private void Dispose(bool disposing)  {  if (!disposed)  {  if (disposing)  {  //Executati eliberarea resurselor  WriteLine("Eliberarea resurselor in dispose");  }  //Eliberarea resurselor critice  disposed = true;  }  }  //Executie  static void Main()  {  FinalizeDemo demo = new FinalizeDemo();  IDisposable dispose;  dispose = demo; //Obtinem referinta spre interfata IDisposable a obiectului dat  dispose.Dispose(); //Apelam metoda Dispose din interfata  }  }  } |

[*Sursa*](https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose?redirectedfrom=MSDN)

După cum vedeți în exemplul prezentat mai sus implementarea metodei Dispose necesită mai multe etape.

1. În primul rând trebuie să ne asigurăm că **clasa care va implementa metoda Dispose va moșteni de la interfața IDisposable**.
2. În al doilea rând **trebuie să implementăm o metodă Dispose fără parametri** în interiorul clasei(acest lucru este obligatoriu din cauza că este cerut de interfață).
3. În final vom utiliza GC.SuppressFinalize pentru a bloca execuția Destructorului**(în caz că acesta există)**, pentru că nu putem distruge un obiect de 2 ori.

Probleme pentru rezolvare.

### **Program 1:**

1. În baza exemplului de mai sus creați 2 clase, una din ele va implementa metoda Dispose și interfața IDisposable, iar cealaltă va crea un obiect din clasa 1 și va apela metoda Dispose din el.