### Modulul 7. Proprietăți și indexatori în C#. Operații Async.

#### Proprietățile în C#.

#### Ce este incapsularea?

**Incapsularea** este procesul protecție a unor date prin setarea nivelului acestora de accesibilitate la o anumită clasă, librărie etc.

Incapsularea poate fi privită ca măsură de prevenție împotriva accesului exterior și modificărilor din surse nedorite, ceea ce subsecvent ajută la minimizarea erorilor logice și creșterea stabilității și structurii codului.

Instrumentul de implementare a incapsulării este nivelul de accesibilitate(public, private etc.).

#### Ce este o proprietate?

După cum am discutat deja de mai multe ori în C# este important să păstrăm variabilele din interiorul claselor private dacă avem posibilitatea pentru a reduce riscul apariției erorilor nedorite din cauza schimbărilor neprevăzute, ceea ce de fapt reprezintă scopul de bază al incapsulării.

Dar există cazuri în care avem nevoie de a accesa aceste variabile din unele clase exterioare.

În acest caz putem utiliza **proprietățile**.

O proprietate este o parte componentă a unei clase care oferă un mecanism simplu de a expune sau de a modifica o variabilă a clasei din exteriorul acesteia. Probabil deja cunoașteți acest mecanism din cuvintele cheie **get** și **set.**

**Get** este folosit pentru a returna/expune o variabilă.

**Set** - pentru a o modifica.

#### Sintaxa declarării unei proprietăți

Pentru a declara o proprietate în C# trebuie să respectăm următoarea sintaxă:

|  |
| --- |
| a) nivelul de accesibilitate b)tipul returnat c) numele proprietății  {  get  set  } |

**Exemplu:**

|  |
| --- |
| class Proprietate  {  private int x = 10;  public int X  {  get{ return x;}  set{ x = value; }  }  }  class Executie  {  static void Main()  {  Proprietate prop = new Proprietate();  WriteLine(prop.X);  prop.X = 5;  WriteLine(prop.X);  }  } |

În exemplul de mai sus, în clasa Proprietate avem o variabilă privată de tip integer x. Aceasta poate fi citită și modificată din interiorul clasei, dar nu din exterior.

Pentru a implementa aceste posibilități am creat o proprietate de tip integer X, care este accesibilă public.

Aceasta returnează valoarea x și permite modificarea acesteia din exterior.

În clasa Executie am folosit proprietatea publică X pentru a schimba valoarea x locală de la 10 la 5 și de a afișa la consolă rezultatul pentru a demonstra funcționalitatea.

**Observație:** se consideră o practică bună de a păstra același nume pentru variabila x și X, dar variabila care va fi accesibilă public va fi notată cu literă mare. Același concept se folosește și în alte cazuri, deși acest lucru nu este obligatoriu, este recomandat.

#### Tipurile proprietăților

* **Proprietate Read/Write:** Proprietate care conține metoda get și set.
* **Proprietate Read-Only(poate fi doar citită):** Este definită doar metoda get.
* **Proprietate Write-Only(poate fi doar modificată, nu citită):** Este definită doar metoda set.
* **Proprietate automată(sau variantă scurtă):** Poate fi read-only, write-only sau read/write în dependență de cum decide programatorul.

**Exemplu:**

|  |
| --- |
| class Proprietate  {  private int x = 10;  public int GetX //Proprietate readonly  {  get{ return x;}  }  public int SetX //Proprietate write only  {  set{ x = value;}  }  public int GetSetX //Proprietate read/write  {  get{ return x;}  set{ x = value;}  }  public int X { get; set;} //Propriete automata  }  class Executie  {  static void Main()  {  Proprietate prop = new Proprietate();  //Obtinem x prin proprietatea readonly  ForegroundColor = ConsoleColor.Red;  WriteLine("Propietate readonly");  WriteLine("Obtine X:{0}", prop.GetX);  //Modificam x prin proprietatea writeonly  ForegroundColor = ConsoleColor.Yellow;  WriteLine("\nPropietate writeonly");  prop.SetX = 6;  WriteLine("X modificat:{0}", prop.GetX);  //Modificam si afisam X prin proprietatea read/write  ForegroundColor = ConsoleColor.Magenta;  WriteLine("\nPropietate read/write");  prop.GetSetX = 20;  WriteLine("X modificat prin read/write:{0}", prop.GetSetX);  //Afisam X prin proprietatea automata apoi il modificam si afisam valoarea noua  ForegroundColor = ConsoleColor.Cyan;  WriteLine("\nPropietate automata");  WriteLine("X initial:{0}", prop.X);  prop.X = 5;  WriteLine("X modificat:{0}", prop.X);  ResetColor();  }  } |

În exemplul de mai sus am expus cum funcționează toate tipurile de proprietăți disponibile în C#. Atrageți atenția că proprietatea automată X inițială are valoarea 0 din cauză că aceasta nu a fost inițializată și nu face referință la valoarea privata x care este 10.

Probleme pentru rezolvare.

### **Program 1:**

### Scrieți un program care veți crea o clasă Persoană cu **variabilele private nume, prenume, cod.**

1. Creați o proprietate read/write în interior ca va permite accesarea și modificarea acestor date.
2. În altă clasă, în metoda Main creați 3 obiecte de tip persoană.
3. Setați și afișați detaliile fiecărei persoane prin intermediul proprietăților.

### **Program 2:**

1. Executați aceeași condiție ca în programul precedent dar de această dată, făceți toate câmpurile din persoană publice cu proprietate automată.

**Exemplu: public string Nume{get; set;}**

#### Indexatorii în C#.

#### Ce este un indexator?

Un indexator este o proprietate specială C# care permite ca o clasă sau structură să fie accesată ca și un tablou de elemente.

Atunci când declarați un indexator în interiorul unei clase/structuri ea va fi considerată un “tablou virtual” și vă permite să accesați și/sau să modificați elementele ei după un anumit index așa ca și cum ați face cu un tabloul unidimensional obișnuit.

Un indexator trebuie declarat în interiorul unei clase sau structuri pentru a funcționa.

#### 

#### Crearea indexatorilor

**Sintaxa indexatorilor:**

|  |
| --- |
| a) nivelul de accesibilitate b)tipul returnat c) cuv cheie this d)index[nr element]  {  get  {  return val[index];  }  set  {  val[index] = value;  }  } |

**Unde putem folosi indexatori și care este importanța lor:**

Utilizarea primară a indexatorilor este de a servi drept metodă ușoară de acces la variabilele unor clase de tip container. Aceștia au fost creați pentru a face codul în C# mai comod și mai intuitiv pentru dezvoltatori.

**Exemplu:**

|  |
| --- |
| using static System.Console;  namespace IntroducereIndexatori  {  class Indexator  {  private string[] valori = new string[3];  public string this[int \_index]  {  get  {  return valori[\_index];  }  set  {  valori[\_index] = value;  }  }  }  class Executie  {  public static void Main()  {  Indexator index = new Indexator();  index[0] = "cuvint1";  index[1] = "cuvint2";  index[2] = "cuvint3";  WriteLine("Valoare 1: {0}", index[0]);  WriteLine("Valoare 2: {0}", index[1]);  WriteLine("Valoare 3: {0}", index[2]);  }  }  } |

În exemplul de mai sus am creat o clasă care va stoca un tabloul de string-uri și va conține un indexator care permite extragerea și schimbarea acestora. Am declarat indexatorul în clasa Indexator și am scris sintaxa corespunzătoare pentru a definit get-ul și set-ul variabilelor din tablou.

În continuare în metoda Main am introdus 3 elemente în tabloul din clasa Indexator, apoi le-am extras, folosind set-ul și get-ul declarat.

Problemă pentru rezolvare.

### **Program 1:**

### Scrieți un program care veți crea o clasă Tranzactii cu **o singură variabilă care va fi un tablou numeric de numere întregi ca va conține suma tranzacțiilor.**

1. Creați un indexator în interiorul clasei pe baza exemplului de mai sus.
2. În altă clasă, în metoda Main creați un obiecte de tip Tranzactii și adăugați câteva valori în tabloul cu sumele tranzacțiilor.
3. Afișați toate tranzacțiile și găsiți tranzacția maximă.

#### Supraîncărcarea indexatorilor

Indexatorii pot fi supraîncărcați pentru a obține o funcționalitate mai largă de la ei. Cea mai des folosită opțiune pentru supraîncărcarea indexatorilor este de a trimite un alt tip de index, nu doar numeric. În acest mod puteți extrage un element de tip string dintr-un tablou de stringuri doar dacă el coincide cu cuvântul pe care îl căutați, sau orice alt exemplu care presupune căutarea unui element de tip X în tablou.

În exemplul de mai jos am creat o clasă IndexatorSupra care conține un tablou de cuvinte ca și în exemplul precedent, dar de data aceasta am declarat 2 indexatori, al doilea fiind varianta supraîncărcată care primește ca index un string și returnează un integer. Funcționalitatea indexatorului 2 este de identifica locul cuvântului primit în tabloul din clasa noastră și de a returna acest loc ca număr.

**Exemplu:**

|  |
| --- |
| class IndexatorSupra  {  static public int size = 4;  private string[] namelist = new string[size];  public string this[int index]  {  get  {  return namelist[index];  }  set  {  namelist[index] = value;  }  }  public int this[string name]  {  get  {  int index = 0;  while (index < size)  {  if (namelist[index] == name)  {  return index;  }  index++;  }  return index;  }  }  static void Main()  {  //Cream obiectul si declaram  IndexatorSupra names = new IndexatorSupra();  names[0] = "Ion";  names[1] = "Ilie";  names[2] = "Igor";  names[3] = "Ivan";  //Printeaza toate componentele tabloului  foreach (string nume in names.namelist) WriteLine(nume);  //Cauta si printeaza locul numelui X  WriteLine("\nLocul numelui cautat este:{0}", names["Igor"]);  }  } |

#### Indexatori multidimensionali

Precum există tablouri multidimensionale, astfel există și indexatori multidimensionali. Conceptul acestora este exact acela care credeți, să vă permită să setați și să extrageți date din tablouri multidimensionale. Acest lucru permite stocarea și modificarea unor structuri de date mai complexe decât masivele unidimensionale.

În exemplul de mai jos am declarat un clasă IndexatorMulti în care vom păstra un masiv bidimensional de numere întregi cu 3 coloane și 3 linii. În continuare am stabilit un indexator care ne permite să setăm și să extragem valori din tablou.

În clasa Executie am creat ca de obicei un obiect de tip IndexatorMulti și am setat valorile tabloului din interior după care am recurs la metoda get din indexator pentru a extrage numerele de care avem nevoie.

**Exemplu:**

|  |
| --- |
| class IndexatorMulti  {  int[,] data = new int[3, 3];  public int this[int index1, int index2]  {  get  {  return data[index1, index2];  }  set  {  data[index1, index2] = value;  }  }  }  class Executie  {  public static void Main()  {  IndexatorMulti index = new IndexatorMulti();  index[0, 0] = 9; index[0, 1] = 5; index[0, 2] = 6;  index[1, 0] = 4; index[1, 1] = 5; index[1, 2] = 6;  index[2, 0] = 7; index[2, 1] = 8; index[2, 2] = 9;  WriteLine("{0}\t{1}\t{2}\n{3}\t{4}\t{5}\n{6}\t{7}\t{8}",  index[0, 0], index[0, 1], index[0, 2],  index[1, 0], index[1, 1], index[1, 2],  index[2, 0], index[2, 1], index[2, 2]);  }  } |

### Modul adițional 1. Operațiile Async.

#### Operații Async în C#.

#### Ce sunt operațiile Async?

Operațiile Async sunt în esență un șir de operații ca se execută într-o anumită ordine și vor aștepta mereu până când operația precedentă e finisată înainte de a începe execuția. Deci spre exemplu dacă avem 3 operații, prima va fi executată operația 1, în acest timp operația 2 va aștepta până când 1 va fi executată apoi va începe și ea execuția.

Avantajul principal al operațiilor Async este că acestea vă pot ajuta să divizați logica în mai multe părți va se vor aștepta una pe alta.

**Exemplu:**

|  |
| --- |
| namespace Async  {  class Program  {  static void Main()  {  Method1();  Method2();  ReadKey();  }  public static async Task Method1()  {  await Task.Run(() =>  {  for (int i = 0; i < 20; i++)  {  Write("1");  Thread.Sleep(100);  }  });  }  public static async Task Method2()  {  await Task.Run(() =>  {  for (int i = 0; i < 20; i++) Write("2");  });  }  }  } |

În exemplul de mai sus am demonstrat ce se va întâmpla dacă două operații async vor fi executate simultan fără a se aștepta una pe alta.

#### Sintaxa

Pentru a crea o operație Async trebuie să respectăm următoarea sintaxă:

|  |
| --- |
| a) Nivelul de acces b) cuv cheie async b)cuv cheie task Task b) numele operației()  {  //Logica  } |

**Observație:** Operațiile async se execută cu întârziere, de aceea trebuie să includeți un mecanism de așteptare, altfel programul se poate închide fără a afișa rezultatul.

**Observație:** Pentru a folosi operațiile Async trebuie să includem librăriile

using System.Threading;

using System.Threading.Tasks;

#### Ordonarea metodelor

În exemplul precedent am afișat cum metodele pot afișa detalii fără a aștepta rezultatul metode precedente, dar avantajul de bază a operațiilor async este anume ceea că putem crea o logică structurată care va aștepta până metodele precedente au fost finisate.

Pentru a obține rezultatul dat vom utiliza cuvântul cheie **await urmat de numele operației pe care o așteptăm să fie finisată.**

**Exemplu:**

|  |
| --- |
| class Program  {  static void Main()  {  MetodaGeneral();  ReadKey();  }  public static async Task MetodaGeneral()  {  await Method3();  await Method2();  await Method1();  await Start();  }  public static async Task Method3()  {  await Task.Run(() =>  {  WriteLine("3");  Beep();  Thread.Sleep(1000);  });  }  public static async Task Method2()  {  await Task.Run(() =>  {  WriteLine("2");  Beep();  Thread.Sleep(1000);  });  }  public static async Task Method1()  {  await Task.Run(() =>  {  WriteLine("1");  Beep();  Thread.Sleep(1000);  });  }  public static async Task Start()  {  await Task.Run(() =>  {  WriteLine("Start!");  Beep(1000, 1000);  });  }  } |

În rezultatul executării programului de mai sus vom obține 4 operații executate în ordine.

Problemă pentru rezolvare.

### **Program 1:**

### Scrieți un program în care veți crea o operație async cu parametru string.

### Operația va afișa fiecare caracter cu o anumită întârziere.

### Creați o operația generală care va apela de mai multe ori operația de afișare a cuvintelor cu diferite cuvinte, în rezultat trebuie să aveți o propoziție care se afișează literă după literă cu întârziere.

|  |
| --- |
| namespace Problema  {  class Program  {  private string cuvint;  static void Main()  {  MetodaGeneral();  ReadKey();  }  public static async Task MetodaGeneral()  {  await Cuv1("Alo!");  await Cuv1(" Da la cine am nimerit?");  await Cuv1(" Ma scuzati atunci...");  }  public static async Task Cuv1(string \_cuv1)  {  await Task.Run(() =>  {  for (int i = 0; i < \_cuv1.Length; i++)  {  Write(\_cuv1[i]);  Beep(500, 100);  Thread.Sleep(100);  }  });  }  }  } |

### Probleme pentru acasă.

### **Program 1:**

### Scrieți un program în care veți crea o **structură** Elev cu variabilele private Nume, Prenume, Nota Medie.

### Creați proprietăți publice pentru fiecare variabilă.

### În altă clasă, în metoda Main creați câteva obiecte de tip Elev și veți afișa informația despre ei literă cu literă(ca în problema rezolvată în timpul orei Async).

### **Program 2:**

### Scrieți un program în care veți crea o **clasă** Container care va conține un tablou de numere integer.

### Creați un indexator în interiorul clasei ca va permite accesarea și modificarea elementelor din tablou.

### În altă clasă, în metoda Main, în interiorul unui ciclu introduceți numere random în tabloul integer din **Container.**

### Cu ajutorul indexatorului extrageți un număr random din tabloul din clasa Container și afișați valoarea.

### **Program 3:**

### Scrieți un program în care veți crea o **clasă** Numere, în care veți introduce un tabloul de numere float.

### Creați un indexator în interiorul clasei ca va permite accesarea și modificarea elementelor din tablou.

### Creați un indexator supraîncărcat care va returna ordinea unui element dacă acesta este găsit în tablou.

### În altă clasă, în metoda Main creați un obiect de tipul Numere.

### Adăugați și afișați elementele din tabloul cu ajutorul indexatorului.

### Găsiți numărul de ordine al unui element din tabloul cu ajutorul indexatorului supraîncărcat.

### Modul adițional 2. Continuare în WPF. Deschiderea fișierelor. Poziții flexibile ale elementelor.

#### Deschiderea fișierelor în WPF

La ora precedentă am început să creăm playerul nostru audio în WPF, următorul pas logic este să adăugăm funcționalitatea de a alege fișierul care vrem să îl deschidem.

Pentru a realiza acest scop vom utiliza libăriile Microsoft.Win32 care conțin clase ajutătoare pentru lucrul cu fișierele Windows.

În cazul nostru vom folosit clasa **OpenFileDialog** care permite deschiderea unei ferestre în care putem alege fișierul de care avem nevoie.

**Codul responsabil pentru deschiderea fișierului și deschiderea cântecului:**

|  |
| --- |
| private void OpenFile\_Click(object sender, RoutedEventArgs e)  {  Microsoft.Win32.OpenFileDialog window = new Microsoft.Win32.OpenFileDialog();  window.Filter = "All Supported Audio | \*.mp3; \*.wma | MP3s | \*.mp3 | WMAs | \*.wma";  Nullable<bool> result = window.ShowDialog();  if (result == true)  {  string filename = window.FileName;  filePath = filename;  OpenSong();  PressPlay();  }  } |

#### Poziții flexibile ale elementelor

Unul din avantajele principale ale WPF este că puteți seta pozițiile elementelor grafice în mod flexibil.

Pentru a obține acest rezultat trebuie să creați mai multe linii de grid, prin apăsarea pe bara de sus sau din părți a ferestrei de design. Acestea vor servi drept puncte de reper pentru elementele grafice.

În continuare puteți plasa elementele relativ cu aceste linii ceea ce va restrânge poziția lor în locul care l-ați stabilit.