### Modulul 11. Evenimente și delegări.

#### Delegări

1. Definiția delegării

Delegările în C# sunt similare cu pointerele spre metode în C și C++. În esență, **o delegare este o referință către o metodă.** Referința dintr-o delegare poate fi modificată în timpul execuției programului din motiv că acesta este o variabilă de tip referință, nu valoare.

Utilizarea cea mai des răspândită a delegărilor este:

1. La implementarea evenimentelor și a callback-urilor.
2. Când vrem să transmitem o metodă ca parametru.

Delegările fac parte din clasa System.Delegate.

1. Sintaxa declarării unei delegări

Pentru a face o delegare să funcționeze trebuie să îndeplinim trei pași:

1. Declarăm delegarea
2. Setăm care metodă o va reprezenta delegarea
3. Apelăm delegarea.

Ca să declarăm o delegare vom urma sintaxa următoare:

|  |
| --- |
| [nivelul de acces] **delegate** [tipul returnat] [numele delegării]([parametri]) |

**Exemplu:**

|  |
| --- |
| namespace Delegare  {  class Delegare  {  public delegate void MyDelegate(string msg);  static MyDelegate del = MethodA;  static void MethodA(string message)  {  WriteLine(message);  }  private static void Main()  {  del("Alo");  }  }  } |

În exemplul de mai sus am declarat o delegare MyDelegate care primește ca parametru un string. Apoi am o nouă delegare de tip MyDelegate și i-am setat metode către MethodA. După aceea în metoda Main a programului putem apela delegarea del ca și cum am apela o metodă obișnuită.

**Exemplu:**

|  |
| --- |
| public delegate void MyDelegate(string msg);  class Program  {  static void Main(string[] args)  {  MyDelegate del = ClassA.MetodaA;  del("Metoda 1");  del = ClassB.MetodaB;  del("Metoda 2");  }  }  class ClassA  {  public static void MetodaA(string message)  {  WriteLine("Apelam metoda A cu mesajul: " + message);  }  }  class ClassB  {  public static void MetodaB(string message)  {  WriteLine("Apelam metoda B cu mesajul: " + message);  }  } |

În exemplul de mai sus am demonstrat același concept ca și în exemplul precedent, dar de data aceasta am utilizat două delegări pentru a apela metode din 2 clase diferite cu diferiți parametri.

Probleme pentru rezolvare.

### **Program 1:**

1. Creați un program în care veți declara o delegare de tip integer.
2. În clasa Execuția, în metoda Main creați două delegări care fac referințe la două metode din clase exterioare(ca în exemplu).
3. Calculați suma cu cifrele pe care le-ați obținut din delegări.

### **Program 2:**

1. Creați un program în care veți declara o delegare de tip string.
2. În clasa Execuția, în metoda Main creați două delegări care fac referințe la două metode din clase exterioare(ca în exemplu).
3. Concatenați primele trei litere din fiecare cuvânt.
4. Scopurile delegării

Delegările oferă mai multe avantaje pentru utilizatorii C#, dar cele mai importante sunt:

1. Permite incapsularea la una sau mai multe metode în interiorul unei singure delegări.
2. Al doilea avantaj este că putem folosi delegările pentru a le transmite ca parametri pentru alte metode, ceea ce permite să executăm anumite funcții doar în cazurile în care avem nevoie(de exemplu atunci utilizator tastează un buton, executăm 3 metode etc.)
3. Delegările permit crearea evenimentelor pe baza lor, ceea ce semnifică că putem crea sisteme care vor urmări execuția metodelor din delegare și vor reacționa când se întâmplă un anumit eveniment.
4. Apelarea mai multor metode prin delegare(multicasting)

Deosebim două tipuri de delegări:

1. Singlecast - care fac referință doar la o metodă
2. Multicasting - care fac referință la mai multe metode

Delegările pot fi adunate și scăzute între ele pentru a obține o consecutivitate de metode care se vor executa în ordine. Delegările care conțin referințe la mai multe metode se numesc Multicasting.

Pentru a efectua operațiile de adiție sau ștergere a unei metode din delegare folosim operațiile obișnuite de +, -, +=, -=.

**Exemplu:**

|  |
| --- |
| namespace Multicasting  {  public delegate void MyDelegate();  class Program  {  static void Main(string[] args)  {  MyDelegate del1 = Class1.Metoda1;  MyDelegate del2 = Class2.Metoda2;  MyDelegate del3 = Class3.Metoda3;  MyDelegate delTotal;  delTotal = del1 + del2 + del3;  delTotal();  }  }  class Class1  {  public static void Metoda1()  {  WriteLine("Metoda 1");  }  }  class Class2  {  public static void Metoda2()  {  WriteLine("Metoda 2");  }  }  class Class3  {  public static void Metoda3()  {  WriteLine("Metoda 3");  }  }  } |

În exemplul de mai sus am folosim adunarea delegărilor pentru a crea o delegare finală care incorporează execuția la toate 3 metode.

Probleme pentru rezolvare.

### **Program 1:**

1. Creați un program în care veți declara o delegare de tip string cu parametru string.
2. Creați 3 metode de tip string cu parametru string:
3. Prima metodă va șterge toate literele ‘a’ din cuvântul parametru.
4. A doua metodă va face toate literele majuscule.
5. A treia metodă va adăuga “xyz” la sfârșitul cuvântului.
6. Creați trei delegări spre fiecare metodă, și a patra care le va implementa pe toate(prin multicasting).
7. Transmiteți un cuvânt parametru și afișați rezultatul.

**Rezolvare: (aici nu era necesară adunarea delegărilor)**

|  |
| --- |
| namespace Problema  {  public delegate string MyDelegate(string msg);  class Program  {  static void Main(string[] args)  {  MyDelegate del1 = ClassA.MethodA;  MyDelegate del2 = ClassB.MethodB;  MyDelegate del3 = ClassC.MethodC;  WriteLine(del3(del2(del1("aleator"))));  }  }  class ClassA  {  public static string MethodA(string msg)  {  return msg.Replace("a", "");  }  }  class ClassB  {  public static string MethodB(string msg)  {  return msg.ToUpper();  }  }  class ClassC  {  public static string MethodC(string msg)  {  return msg + "xyz";  }  }  } |

### 

|  |
| --- |
| namespace Problema  {  public delegate string MyDelegate(ref string msg);  class Program  {  static void Main(string[] args)  {  string cuvint = "aleator";  MyDelegate del1 = ClassA.MethodA;  MyDelegate del2 = ClassB.MethodB;  MyDelegate del3 = ClassC.MethodC;  MyDelegate delTotal = del1 + del2 + del3;  delTotal(ref cuvint);  WriteLine(cuvint);  }  }  class ClassA  {  public static string MethodA(ref string msg)  {  return msg = msg.Replace("a", "");  }  }  class ClassB  {  public static string MethodB(ref string msg)  {  return msg = msg.ToUpper();  }  }  class ClassC  {  public static string MethodC(ref string msg)  {  return msg += "xyz";  }  }  } |

### 

### **Program 2:**

1. Creați un program în care veți declara o delegare de tip void fără parametri.
2. Creați 3 metode de tip void:
3. Prima afișa patratul la un număr.
4. A doua metodă va afișa o literă aleatori.
5. A treia metodă va face un sunet la consolă
6. Creați trei delegări spre fiecare metodă, și a patra care le va implementa pe toate(prin multicasting).
7. Invocați delegarea finală.
8. Delegări ca parametri pentru metode

După cum am spus mai devreme delegărilor pot fi transmise și ca parametri pentru metode, ceea ce ne permite să integrăm orice funcție dorim în interiorul altor funcții.

Avantajul acestei posibilități este că putem crea metode generice care vor permite execuția oricăror alte metode în interiorul acestora. Adițional putem folosi delegări ca parametri împreună cu multicasting pentru a crea secvențe întregi de metode.

**Exemplu:**

|  |
| --- |
| namespace DelegariParametru  {  public delegate void MyDelegate();  class Buton  {  public static Random rand = new Random();  static void Main()  {  MyDelegate[] del = new MyDelegate[]{ Text.Rezultat, Calcul.Rezultat, Sunet.Rezultat };  int num = rand.Next(3);  ClickButon(del[num]);  }  public static void ClickButon(MyDelegate \_delegate)  {  \_delegate();  }  }  class Text  {  public static void Rezultat()  {  WriteLine("Am primit rezultat un text.");  }  }  class Calcul  {  public static void Rezultat()  {  WriteLine(5+5);  }  }  class Sunet  {  public static void Rezultat()  {  Beep(500, 500);  }  }  } |

Pentru a demonstra argumentul de mai sus am creat programul de mai sus în care clasa Buton conține metoda ClickButon care primește o delegare ca argument.

În rezultat Butonul nostru poate executa o multitudine de funcții în dependență de argumentul pe care îl primește. Ca să finisăm problema am creat un masiv de delegări, în care am inclus mai multe referințe către metodele din clasele Text, Calcul și Sunet. Pentru a alege aleatoriu una din aceste metode am creat un număr random de la 0 până la 2, în dependeță de acest număr vom executa o metodă diferită în interiorul metodei ClickButon.

Probleme pentru rezolvare.

### **Program 1:**

1. Creați un program în care veți avea o clasă de bază Angajat.
2. În interiorul clasei creați o metodă InteractiuneAngajat cu parametru delegare.
3. Creați 3 clase exterioare, Promovare, Concediere, Nimic. În interiorul fiecăreia creați câte o metodă care va afișa un mesaj corespunzător la consolă.
4. În clasa de bază creați un tablou de delegări, cu referințe către toate metodele din clasele Promovare, Concediere, Nimic.
5. În metoda Main creați o variabilă int aleatorie de la (0 până la 100).
6. Dacă variabila este:
7. Mai mare ca 75, apelați metoda Promovare
8. Dacă între 50 și 75, apelați metoda Nimic.
9. Dacă mai mică de 50 metoda Concediere.

#### Evenimente

1. Ce este un eveniment în C#?

Un eveniment în C# reprezintă o notificare transmisă de un obiect că o anumită acțiune a fost îndeplinită.

Când vorbim de evenimente în C# includem 2 entități incluse:

1. Obiectul care crează evenimentul care se numește Publisher(publică evenimentul).
2. Obiectul care primește evenimentul, care se numește Subscriber(primește notificare).

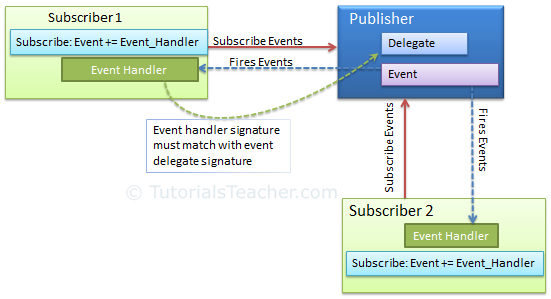
De obicei Publisher-ul crează un eveniment când o anumită acțiune a fost îndeplinită, iar Subscriberii urmăresc când acest lucru se întâmplă și reacționează prin execuția unei anumite logici.

Sistemul dat este numit **Publisher-Subscriber** model sau **Observer Design Pattern** din motiv că se focusează pe crearea entităților care reacționează doar atunci când este nevoie, dar nu execută permanent verificări.

Evenimentele în C# depind direct de delegări, ceea ce vom observa imediat în exemplele ce urmează, dar acestea pot folosi de asemenea și clasa EventHandler.

Evenimentele se folosesc pentru a denota apariția unei acțiuni care necesită prelucrare, precum apăsarea unei taste, a mouse-ului, terminarea unui proces etc.

**Cum funcționează un eveniment în C#.**



1. Sintaxa evenimentelor

Un eveniment în C# reprezintă o notificare transmisă de un obiect că o anumită acțiune a fost îndeplinită.

Pentru a declara un eveniment în C# trebuie să declarăm o delegare întâi. Apoi după ce delegarea a fost declarată în interiorul clasei care va genera evenimente creăm un eveniment cu ajutorul cuvântului cheie **event**.

**Sintaxa:**

|  |
| --- |
| public delegate void MyDelegate();  class Event  {  public event MyDelegate ProcesFinisat;  } |

În cazul dat evenimentul ProcesFinisat are tipul delegării MyDelegat

1. Utilitatea și specificul utilizării evenimentelor

Utilitatea principală a utilizării evenimentelor este că acestea permit programatorilor să creeze o consecutivitate logică de operații logice în timp ce maximizează performanța programului. Din motiv că modelul Publisher-Subscriber este unul pasiv(subcriberii nu executa nici o verificare și nici o logică până când publisher-ul nu crează un eveniment), acest lucru ajută la majorarea semnificativă a permormanței programului.

**Exemplu:**

|  |
| --- |
| namespace Eveniment  {  public delegate string MyDel(string str);  class EventProgram  {  event MyDel MyEvent;  public EventProgram() { MyEvent += new MyDel(WelcomeUser); }  public string WelcomeUser(string username) { return "Rezultat: " + username; }  private static void Main()  {  EventProgram obj1 = new EventProgram();  string result = obj1.MyEvent("Eveniment creat.");  WriteLine(result);  }  }  } |

În exemplul de mai sus am utilizat delegarea MyDel pentru a crea un eveniment pe baza ei, de fiecare dată când va fi utilizat constructorul clasei EventProgram, vom genera un eveniment MyEvent, care va conține un mesaj string.

Deși acest exemplu demonstrează cum se crează un eveniment aici avem doar o clasă publisher, dar nu există clase subscriberi.

**Exemplu:**

|  |
| --- |
| namespace EvenimentPublisherSubscriber  {  public delegate void MyDel(bool \_status);  public class Publisher  {  public event MyDel ProcessCompleted;  public void StartProcess()  {  try  {  WriteLine("Proces Inceput");  OnProcessCompleted(true);  }  catch (Exception ex) { OnProcessCompleted(false); }  }  protected void OnProcessCompleted(bool IsSuccessful)  {  ProcessCompleted?.Invoke(IsSuccessful);  }  }  class Subscriber  {  public static void Main()  {  Publisher publisher = new Publisher();  publisher.ProcessCompleted += subscriber\_ProcessCompleted;  publisher.StartProcess();  }  public static void subscriber\_ProcessCompleted(bool IsSuccessful)  {  WriteLine("Ajuns aici.");  WriteLine("Proces " + (IsSuccessful ? "finisat cu succes." : "esuat."));  }  }  } |

Scopul exemplului de mai sus este să demonstreze cum interacționează un eveniment din clasa Publisher cu clasa Subscriber. Am declarat ca, de obicei o delegare pe bază căreia am creat un eveniment ProcessCompleted în clasa Subscriber. În continuare am declarat o funcție publică StartProcess care va fi apelată dintr-o clasă exterioară pentru va începe procesul.

Când vom începe procesul vom afișa la consolă Proces Început, și dacă procesul este finisat cu succes vom invoca metodele din evenimentul ProcessCompleted.

**Detalii importante:**

* În C# evenimentele se folosesc pentru a trimite notificări unor clase abonate că o anumită operație a fost executată în clasa publisher.
* Pentru a declara un eveniment avem nevoie de o delegare.
* Folosind operatorul += putem abona un obiect la un eveniment, folosing -= îl putem dezabona.
* Un singur eveniment poate avea mai mulți subscriberi, și un singur subscriber poate fi abonat la mai multe evenimente.
* Publisher-ul alege când un eveniment este inițializat, dar obiectele abonat aleg cum să prelucreze evenimentul.

Probleme pentru rezolvare.

### **Program 1:**

1. Pe baza exemplului de mai sus, creați un program în care veți avea 2 clase, ContBancar(care va fi publisher) și Tranzactie(care va fi subscriber).
2. În clasa ContBancar declarați o variabilă int balantaContului care va conține toate mijloacele bănești din cont.
3. În aceeași clasă creați o metodă InteractiuneCuMijloace(int \_modif) care va primi un parametru int, această metoda va adăuga sau retrage mijloace bancare din cont.
4. În clasa Tranzacție, creați o metodă cu parametru int care va adăuga și retrage mijloace din variabila balantaContului din clasa ContBancar.
5. Atașați metoda la eveniment și încercați să executați.

**Observație:**Tranzacția este imposibilă dacă balanța contului este mai mică de 0.

#### Evenimente(continuare)

1. Exemple și utilizare

Pentru a adăuga o metodă la un eveniment este suficient să folosim un operator aritmetic + sau +=, și pentru a șterge o metodă din eveniment facem operația inversă -, -=.

**Exemplu:**

|  |
| --- |
| namespace EvenimenteContinuare  {  public class MyTest  {  public delegate void MyDel();  public event MyDel MyEvent;  public void StartMethod()  {  MyEvent?.Invoke();  }  }  public class Test  {  public void TestEvent()  {  MyTest myTest = new MyTest();  myTest.MyEvent += myTest\_MyEvent;  myTest.StartMethod();  myTest.MyEvent -= myTest\_MyEvent;  myTest.StartMethod();  }  public void myTest\_MyEvent()  {  Write("X");  }  static void Main(string[] args)  {  Test test = new Test();  test.TestEvent();  }  }  } |

Probleme pentru rezolvare.

### **Program 1:**

1. Pe baza exemplului de mai sus, creați un program în care veți avea 3 clase, Publisher(care va fi conține evenimentul) și Subscriber(care va fi trimite delegări) și execuție unde veți executa programul.
2. În clasa Subscriber creați un constructor cu parametru string, și o metodă care va afișa la consolă acest string. De asemenea în constructor atașați metoda dată la evenimentul din Publisher.
3. În clasa execuție creați un obiect de tip Publisher și câteva de tip Subscriber.
4. Invocați evenimentul.

**Exemplu 2:**

|  |
| --- |
| using static System.Console;  namespace Evenimente  {  class Executie  {  private static void Main()  {  RequestServer newRequest = new RequestServer();  RequestServer.RequestEvent += Sunet.PlaySound;  RequestServer.RequestEvent += MesajConsola;  newRequest.TrimiteRequest();  }  private static void MesajConsola()  {  ForegroundColor = System.ConsoleColor.Red;  WriteLine("Request finisat");  }  }  }  using System;  using System.Threading;  namespace Evenimente  {  class Sunet  {  public static void PlaySound()  {  Console.Beep(500, 200);  Thread.Sleep(3000);  }  }  }  using System;  using System.Net;  using System.IO;  using System.Threading.Tasks;  using static System.Console;  namespace Evenimente  {  //1 - Cream delegarea  public delegate void RequestServerDelegate();  class RequestServer  {  //2 - Cream evenimentul  public static event RequestServerDelegate RequestEvent;  public static StreamReader serverResponse;  public static string urlAddress = "http://interflix.me";  private static void OnRequestFinished()  {  RequestEvent?.Invoke();  }  public void TrimiteRequest()  {  var t = Task.Run(() => SendRequest(urlAddress));  t.Wait();  var z = Task.Run(() => PrintData(serverResponse));  z.Wait();  //3 - Indicam cind evenimentul se apeleaza  OnRequestFinished();  }  #region ServerRequest  private static async Task SendRequest(string \_url)  {  await Task.Run(() =>  {  HttpWebRequest request = (HttpWebRequest)WebRequest.Create(\_url);  HttpWebResponse response = (HttpWebResponse)request.GetResponse();  Stream receiveStream = response.GetResponseStream();  StreamReader readStream = null;  readStream = new StreamReader(receiveStream);  serverResponse = readStream;  });  }  private static async Task PrintData(StreamReader \_data)  {  await Task.Run(() =>  {  string raspuns = \_data.ReadToEnd();  Write(raspuns+"\n");  });  }  #endregion  }  } |

Probleme pentru rezolvare.

### **Program 1:**

1. Creați un program care va avea două clase, Execuție și Așteptare.
2. În clasa Așteptare creați o delegare și un eveniment.
3. Tot în clasa așteptare creați o metodă va executa o operație, aștepta X secunde, și va face o altă operație. La final va invoca Evenimentul.
4. În clasa execuție creați un obiect de tip așteptare și chemați metoda din interior.
5. Practici utile pentru utilizarea evenimentelor

Există mai multe metode care vă pot ajuta să efectuați operații adiționale cu evenimentele.

1. În cazul în care aveți nevoie de numărul de delegări în eveniment și sau numele acestora.

|  |
| --- |
| namespace EvenimentePracticiUtile  {  public delegate void EventHandler();  class Program  {  public static event EventHandler eveniment;  static void Main()  {  eveniment += ClickItem;  eveniment += ClickItem;  eveniment += ClickItem;  //Afiseaza toate delegarile din eveniment  foreach (var item in eveniment.GetInvocationList())  WriteLine(item.Method.Name);  }  static void ClickItem(){}  }  } |

Probleme pentru rezolvare.

### **Program 1:**

1. Folosiți programul de mai sus dar creați 2 metode în loc de una.
2. În metoda Main creați două cicluri care vor atașa X și Y metode aleatorii la evenimentul 1 și 2.
3. Afișați numărul de evenimente din evenimentul 1 și 2, și suma acestora.
4. Precum puteți folosi proprietățile pentru a proteja variabilele puteți adăuga add și remove la un eveniment. Acestea se vor executa atunci când evenimentul vostru va primi o delegare nouă.

|  |
| --- |
| namespace EvenimentePracticiUtile2  {  public delegate void EventHandler();  class Program  {  private static EventHandler evenimentIntermediar;  public static event EventHandler eveniment  {  add { WriteLine("Eveniment adaugat"); evenimentIntermediar += value; }  remove { WriteLine("Eveniment sters"); evenimentIntermediar -= value; }  }  public static EventHandler event2;  static void Main()  {  eveniment += ClickItem;  eveniment += ClickItem;  eveniment += ClickItem;  eveniment -= ClickItem;  eveniment -= ClickItem;  evenimentIntermediar();  }  static void ClickItem() { WriteLine("X"); }  }  } |

Atrageți atenția că în acest caz va trebui să folosim un eveniment intermediar pentru a inițializa evenimentul.

#### Metode anonime

După cum numele sugerează, metodele anonime sunt metodele care nu au un nume. Metodele anonime în C# pot fi create cu ajutorul delegărilor și conțin în ele un bloc de logică declarat de programator.

**Exemplu:**

|  |
| --- |
| namespace MetodeAnonime  {  class Executie  {  public delegate void Print(int value);  static void Main()  {  Print print = delegate (int val){ WriteLine("Metoda Anonima. Valoarea: {0}", val); };  print(25);  }  }  } |

Pentru a declara metoda anonimă am creat o delegare de tip Print, în Main(), apoi am declarat logica care trebuie executată imediat după.

Acest lucru ne poate ajuta când avem nevoie să declarăm unele metode în interiorul altor metode și să le reutilizăm de câteva ori.

O altă utilitate a metodelor anonime este că acestea pot fi transmise altor delegări care pot primi ca parametru o delegare. Ceea ce poate suna confuz, dar în esență este simplu și poate ajuta la crearea unor metode foarte abstracte și flexibile.

**Exemplu:**

|  |
| --- |
| namespace MetodeAnonimeParametru  {  public delegate void Print(int value);  class Executie  {  public static void PrintHelperMethod(Print printDel, int val)  {  val += 10;  printDel(val);  }  static void Main()  {  PrintHelperMethod(delegate (int val) { WriteLine("Metoda anonima. Rezultat: {0}", val); }, 100);  }  }  } |

În exemplul dat am început problema cu declararea unei delegări Print cu parametru int. În clasa execuție am creat o metodă statica PrintHelperMethod, care primește 2 parametri: o delegare și un număr int. Numărul se incrementează cu 10, iar delegarea este executată apoi.

Apoi pentru a implementa metodele anonime, în metoda Main am apelat funcția PrintHelper() dar în loc să creăm o delegare nouă, am folosit o metodă anonimă declarată direct în locul unde trebuie să includem parametrul funcției.

În rezultat, am trimis numărul 100, care va fi incrementat cu 10, apoi metoda anonimă pe care am transmis-o va afișa"Metoda anonima. Rezultat: 110" la consolă. Această tehnică are avantaje semnificative din motiv că putem crea o metodă centralizată care să primească logică diferită de la diverse clase periferice, ceea ce va permite execuția unui număr imens de comenzi dintr-o singură metodă.

Probleme pentru rezolvare.

### **Program 1:**

1. Creați un program în care veți crea o delegare cu doi parametri, un integer și un string.
2. Creați o metodă care va primi 3 parametri(o metodă, un integer și un string), numărul integer va fi ridicat la puterea 3 și divizat la 2, iar cuvântul string va fi dublat.
3. Folosiți metoda parametru pentru a afișa la consolă “Rezultat: {int} {string}.”

#### Expresii Lambda

Expresiile Lambda sunt similare cu metodele anonime dar reprezintă o formă mai scurtă a acestora.

**Metodă anonimă:**

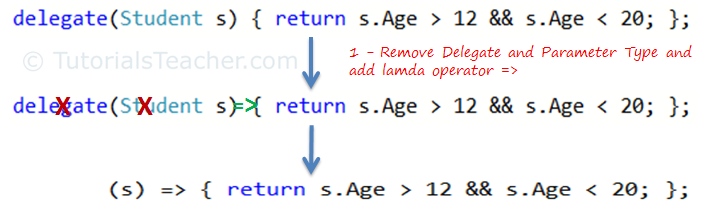
|  |
| --- |
| delegate(Student s) { return s.Age > 12 && s.Age < 20; }; |

**Expresie Lambda:**

|  |
| --- |
| s => s.Age > 12 && s.Age < 20 |

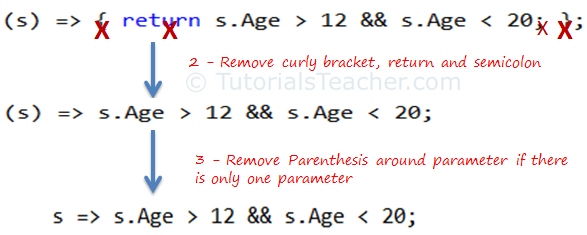
**Structura unei expresii lambda:**

Expresiile lambda au evoluat de la metodele delegate prin simplificarea cuvântului cheie delegate, a denumirii delegării și prin introducerea operatorului terțiar.



Sursă: <https://www.tutorialsteacher.com/linq/linq-lambda-expression>

Expresia de mai sus poate fi simplificată și mai mult prin scoaterea acoladelor și a cuvântului cheie return.



Utilizarea expresiilor lambda:

Expresiile lambda pot fi folosite pentru înlocuirea metodelor anonime mai ales atunci când avem nevoie să transmitem o metodă ca parametru.

**Exemplu:**

|  |
| --- |
| namespace ExpresiiLambda1  {  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);  });  }  }  } |

Să ne întoarcem la exemplul care l-am trecut deja la lecția 7. În acest program am utilizat metoda Task.Run() pentru a executa o anumită logică într-o metodă async, dar ceea ce nu am discutat la moment este faptul că parametrul care l-am trasmis funcției Run, este de fapt o expresie Lambda.

|  |
| --- |
| public static async Task Start()  {  await Task.Run(**() =>**  **{**  **WriteLine("Start!");**  **Beep(1000, 1000);**  **}); //Expresie Lambda** |

În fragmentul de mai sus observăm sintaxa exactă a unei expresii Lambda, și anume parantezele rotunde care denotă o metodă, urmate de operatorul terțiar, apoi un bloc de cod delimitat cu acolade care conține logica care se va executa.

**Exemplu:**

|  |
| --- |
| namespace ExpresiiLambda2  {  class Program  {  static void Main()  {  //Declaram o lista de numere int  List<int> numbers = new List<int>() { 36, 71, 12, 15, 29, 18, 27, 17, 9, 34 };  Write("Lista numerelor: ");  foreach (var value in numbers) Write("{0} ", value);  WriteLine();  //Calculam patratul numerelor  var square = numbers.Select(x => x \* x);  Write("Patratul numerelor: ");  foreach (var value in square) Write("{0} ", value);  WriteLine();  //Determinam numerele divizibile cu 3  List<int> divBy3 = numbers.FindAll(x => (x % 3) == 0);  Write("Numerele divizibile cu 3: ");  foreach (var value in divBy3) Write("{0} ", value);  WriteLine();  }  }  } |

În exemplul prezentat mai sus am demonstrat încă o aplicație a expresiei Lambda. De data aceasta am utilizat expresiile Lambda în combinație cu metodele Select și FindAll din librăria LINQ. În esență, am creat o listă de numere, apoi al calculat patratul acestor numere și am identificat care sunt numerele divizibile cu 3.

Probleme pentru rezolvare.

### **Program 1:**

1. Pe baza exemplului de sus creați un program în care veți inițializa 2 liste numerice.
2. Creați a 3-a listă care le va combina pe primele două.
3. Identificați numărul maxim impar din lista 3.
4. Identificați numărul minim mai mare ca 0 din lista 3.