### Modulul 20. Subiecte adiționale.

#### Clasa Func și expresiile Lambda

Sintaxele de tip Func se întâlnesc des în C# mai ales când utilizăm request-uri LINQ. Func mai este cunoscut și ca “expresii Lambda care returnează o valoare”.

După funcționalitate sintaxa Func are multe similarități cu delegările(și expresiile Lambda evident).

Sintaxa unui Func arată astfel:

|  |
| --- |
| Func<int, string> func1 = (x) => x.ToString(); |

În exemplul de mai sus avem o sintaxă Func care primește ca parametru de intrare un int, pe care îl vom nota cu x și va returna un string, care în cazul dat este doar numărul x convertit în string.

Să analizăm un **exemplu** mai complex.

|  |
| --- |
| namespace ExpresiiFunc  {  class Program  {  static void Main()  {  //Func care primeste un numar si returneaza un string formatat  Func<int, string> func1 = (x) => string.Format("string = {0}", x);  //Func care primeste 2 parametri bool si int si returneaza string  Func<bool, int, string> func2 = (x, y) => string.Format("string = {0} and {1}", x, y);    // Func fara parametri care returneaza Pi/2  Func<double> func3 = () => Math.PI / 2;  Console.WriteLine(func1.Invoke(5));  Console.WriteLine(func2.Invoke(true, 10));  Console.WriteLine(func3.Invoke());  }  }  } |

În exemplul de mai sus avem 3 expresii Func:

1. Prima dintre ele primește un număr int și returnează un string formatat.
2. A doua primește un bool și un int și returnează un string formatat.
3. A treia nu primește parametri dar returnează mereu valoarea Pi/2.

Probleme pentru rezolvare.

### **Program 1:**

1. Pe baza exemplului de mai sus elaborați un program în care veți avea 3 expresii Func:
2. Prima care primește un string și returnează doar prima jumătate a cuvântului.
3. A doua care primește un număr int și returnează numărul dat la puterea 3.
4. A treia primește un număr, dacă acest număr este par, returnați valuarea lui / 2, dacă impar valoarea lui \* 2.

#### Expresii Func ca parametri

După cum am văzut expresiile Func pot fi folosite individual pentru înlocui unele metode sau delegări, dar utilizarea cea mai largă a acestora este în scop de parametri pentru anumite metode, mai ales, dar nu limitat, la cele din LINQ.

În aceste cazuri nu avem nevoie să indicăm că expresia pe care o scriem este de tip Func, vom scrie doar sintaxa care returnează un rezultat.

**Exemplu:**

|  |
| --- |
| namespace ExpresiiFuncParametru  {  class Program  {  static void Main()  {  IEnumerable<string> strings = new List<string> { "one", "two", "three", "four" };  IEnumerable<string> result = strings.Select(str => "Cuvint:" + str);  foreach (var element in result)  Console.WriteLine(element);  }  }  } |

În exemplul dat am creat o colecție IEnumerable de cuvinte, expresia Func pe care am creat-o va avea scopul de a returna un alt string care va adăuga “Cuvint:” înainte de cuvintele pe care le-am declarat în listă.

Pentru a realiza scopul dat și a stoca rezultatele, vom folosi metoda Select care va primi ca parametru o expresie Func.

După cum vedeți în cazul dat expresia Func are scopul de returna o anumită valoare, care apoi va fi plasată în colecția result.

Probleme pentru rezolvare.

### **Program 1:**

1. Pe baza exemplului de mai sus elaborați un program care va returna lungimea cuvintelor din colecția strings.

Probleme pentru rezolvare.

### **Program 2:**

1. Pe baza exemplului de mai sus elaborați un program care va verifica dacă prima literă a cuvântului este ‘t’, dacă da dublați cuvântul(concaținați o copie a lui la sfârșit), dacă nu lăsați cuvântul cum era.

#### Pattern-uri de design al codului în C#

Pattern-uri sau șabloane sunt modele care le întâlnim în multiple cazuri în mai multe programe care se folosesc cu un anumit scop. Cu timpul aceste pattern-uri au devenit standardizate și în prezent există convenții larg acceptate de execuție a acestor șabloane.

Din punct de vedere al scopului, există 3 grupuri de bază de pattern-uri:

1. **De creare (Creational Pattern)**

|  |  |
| --- | --- |
| **Pattern** | **Obiectiv** |
| [**Abstract Factory**](https://www.dofactory.com/net/abstract-factory-design-pattern) | **Reprezintă o clasă care va fi responsabilă pentru crearea altor clase** |
| [**Factory Method**](https://www.dofactory.com/net/factory-method-design-pattern) | **Crează referințe către mai multe clase derivate** |
| [**Prototype**](https://www.dofactory.com/net/prototype-design-pattern) | **Crează o copie inițializată a unui obiect deja existent** |
| [**Singleton**](https://www.dofactory.com/net/singleton-design-pattern) | **Crează un obiect static care nu poate avea replici și poate fi accesat ușor de alte obiecte** |

1. **De structură (Structural Pattern)**

|  |  |
| --- | --- |
| **Pattern** | **Obiectiv** |
| [**Adapter**](https://www.dofactory.com/net/adapter-design-pattern) | **Servește drept punct de unire între două clase diferite** |
| [**Composite**](https://www.dofactory.com/net/composite-design-pattern) | **Crează o structură de tip copac, care împarte un obiect complex în bucăți mai mici care pot fi gestionate individual fără a afecta obiectul de bază** |
| [**Facade**](https://www.dofactory.com/net/facade-design-pattern) | **Un singur obiect care face referință la un întreg subsistem** |

1. **De comportament (Behavioral Pattern)**

|  |  |
| --- | --- |
| **Pattern** | **Obiectiv** |
| [**Iterator**](https://www.dofactory.com/net/iterator-design-pattern) | **Crează o clasă specifică care vă permite să iterați printr-o colecție de obiecte/date într-un mod specific** |
| [**Observer**](https://www.dofactory.com/net/observer-design-pattern) | **O metodă de a notifica un număr mare de obiecte de un anumit eveniment fără necesitatea de a crea referințe directe sau verificări logice.** |
| [**Strategy**](https://www.dofactory.com/net/strategy-design-pattern) | **Incapsulează un tip de obiecte în interiorul unei clase care permite apelarea tuturor obiectelor** |

#### Atributele în C#

Un atribut în C# reprezintă un tag declarativ care conține informație despre caracteristicile unui element(clasă, structură, enumerare etc.) Această informație caracteristică se adăugă în C# cu ajutorul unui tag[].

**Sintaxa unui atribut:**

|  |
| --- |
| [Serializable]  class MainApp  {  //  } |

Sintaxa de mai sus, precum am discutat la lecția precedentă permite unei clase să fie serializabilă, acest lucru se îndeplinește cu ajutorul atributului [Serializable].

Atributele predefinite în C#:

1. **[Serializable] [NonSerialized]** - definește dacă elementul va fi serializat sau nu.
2. **[Obsolete]** - definește o clasă/metodă/enumerare etc. care este funcțională dar deja învechită, din acest motiv trebuie evitată. De obicei folosim atributul Obsolete în cadrul unor programe cu arhitectură mai complexă, cu mai mulți programatori pentru a-i informa că este necesar un update.

**Exemplu:**

|  |
| --- |
| namespace Atribute  {  class Program  {  static void Main()  {  // Warning: 'Program.Test()' is obsolete  Test();  }  [Obsolete]  static void Test(){}  }  } |

Când veți utiliza proprietatea Obsolete pentru a defini metoda Test(), atunci compilatorul va crea un Warning de fiecare dată când veți invoca metoda dată.

Probleme pentru rezolvare.

### **Program 1:**

1. În baza programului de mai sus creați 2 metode care execută împărțirea a două numere.
2. Prima va avea atributului Obsolete, aceasta nu va avea try catch implementat.
3. A doua va avea un try catch integrat.
4. Folosiți ambele metode și afișați rezultatul.
5. **[WebMethod] -** atributul WebMethod în esență delimitează un element care poate fi utilizat online

|  |
| --- |
| namespace Atribute  {  [WebService(Namespace = "http://tempuri.org/")]  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]  // To allow this Web Service to be called from script, using ASP.NET AJAX, uncomment the following line.  [System.Web.Script.Services.ScriptService]  public class UtilityWebService : System.Web.Services.WebService  {  public UtilityWebService()  {  //Uncomment the following line if using designed components  //InitializeComponent();  }  [WebMethod]  public string HelloWorld()  {  return "Hello World";  }  [WebMethod]  public int addition(int a, int b)  {  return a + b;  }  }  } |

Mai multă informație despre WebServices: <https://www.c-sharpcorner.com/UploadFile/84c85b/net-web-services/>

1. **[DLLImport] -** permite importarea unei librării DLL specifice.

|  |
| --- |
| namespace Atribute  {  public class Test  {  [DllImport("user32.dll", EntryPoint = "MessageBox")]  public static extern int ShowMessageBox(int hWnd, string text, string caption, uint type);  }  class Program  {  static void Main()  {  string caption = "Hello World";  string text = "Sample Article on DLLImport Attribute";  Test.ShowMessageBox(0, text, caption, 0);  }  }  } |

În exemplul de mai sus am folosit atributul DllImport pentru a include librăria user32.dll în programul nostru. Acest lucru ne permite să afișăm un mesaj standard Windows cu un anumit mesaj prestabilit.

Probleme pentru rezolvare.

### **Program 1:**

1. Folosiți exemplul de mai sus și modificați fereastra să conțină header-ul SPAM, și mesajul Acesta este spam.
2. Folosiți un ciclul for pentru a afisa 10 ferestre.
3. Măriți numărul de ferestre la 1000 și trimiteți programul unui prieten :)

Atributele specifice(care le puteți crea singuri):

1. **Simple** - acestea sunt atributele care pot fi create de către programator, iar apoi atașate de unele elemente.

**Exemplu:**

|  |
| --- |
| namespace AtributeCustom  {  [AttributeUsage(AttributeTargets.Class)]  public class PerlsAttribute : Attribute {}  [Perls]  class Example1 {}  [PerlsAttribute]  class Example2 {}  } |

În exemplul de mai sus demonstrăm cum pot fi create atribute custom. În primul rând trebuie să creăm o clasă care include atributului **[AttributeUsage()]** și să specificăm pentru care tip de elemente se folosește.

În continuare trebuie să derivăm clasa dată de la clasa System.Attribute.

După aceasta putem să folosim atributul Perls pentru a defini clase.

Atrageți atenția că puteți folosi 2 variante: scurta - Perls, și lungă - PerlsAttribute. Ambele sunt funcționale și se referă la aceeași clasă.

Atributele custom pot fi folosite pentru a valida un anumit tip de date și a verifica dacă unele condiții se îndeplinesc sau nu.

Exemplu în care se folosește validarea cu ajutorul atributelor:

<https://odetocode.com/blogs/scott/archive/2011/02/21/custom-data-annotation-validator-part-i-server-code.aspx>

1. **Multiple** - În cazuril în care avem nevoie să folosim mai multe atribute pentru a specifica sau transmite mai multă informație atunci putem folosi atributele multiple.

**Exemplu:**

|  |
| --- |
| namespace AtributeCustomMultiple  {  [AttributeUsage(  AttributeTargets.Class |  AttributeTargets.Constructor |  AttributeTargets.Field |  AttributeTargets.Method |  AttributeTargets.Property,  AllowMultiple = true)]  public class DeBugInfo : Attribute  {  private int bugNo;  private string developer;  private string lastReview;  public string message;  public DeBugInfo(int bg, string dev, string d)  {  this.bugNo = bg;  this.developer = dev;  this.lastReview = d;  }  public int BugNo  {  get { return bugNo; }  }  public string Developer  {  get { return developer; }  }  public string LastReview  {  get { return lastReview; }  }  public string Message  {  get { return message; }  set { message = value; }  }  }  [DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]  [DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]  class Rectangle  {  protected double length;  protected double width;  public Rectangle(double l, double w)  {  length = l;  width = w;  }  public double GetArea()  {  return length \* width;  }  public void Display()  {  Console.WriteLine("Length: {0}", length);  Console.WriteLine("Width: {0}", width);  Console.WriteLine("Area: {0}", GetArea());  }  }  class ExecuteRectangle  {  static void Main(string[] args)  {  Rectangle r = new Rectangle(4.5, 7.5);  r.Display();  Type type = typeof(Rectangle);  foreach (Object attributes in type.GetCustomAttributes(false))  {  DeBugInfo dbi = (DeBugInfo)attributes;  if (null != dbi)  {  Console.WriteLine("Bug no: {0}", dbi.BugNo);  Console.WriteLine("Developer: {0}", dbi.Developer);  Console.WriteLine("Last Reviewed: {0}", dbi.LastReview);  Console.WriteLine("Remarks: {0}", dbi.Message);  }  }  }  }  } |

Să considerăm exemplul de mai sus. În cazul dat avem o clasă atribut DebugInfo care include numărul bug-ului, numele programatorului, ultima verificare și mesajul erorii.

În cazul dat am indicat AttributUsage poate avea valori multiple pentru a putea fi folosi atât pentru clase, cât și pentru metode, structuri etc, în final trebuie să indicăm **AllowMultiple = true** pentru a permite atributului să includă mai multe elemente.

În continuare clasa Rectangle a primit două atribute care includ toate informațiile menționate mai sus.

În metoda Main cu ajutorul reflecției putem extrage toate atributele atașate la clasa noastră, precum și la alte elemente dacă acestea existau.

În final am afișat la consolă toate atributele.

Probleme pentru rezolvare.

### **Program 1:**

1. Modificați exemplul de mai sus astfel încât atributul să caracterizeze câteva metode(nu clase).
2. Afișați cu ajutorul atributelor numele metodei, clasa din care face parte și timpul exact când a fost invocată.

#### Folosirea GIT. GitHub. Concepte de bază și lucru în echipă.

GIT este o versiune open-source de version control și a fost creat de Linus Torvalds - creatorul Linux.

Deci ce este version control și de ce avem nevoie de el? În timpul creării soft-urilor vom face o mulțime de schimbări în cod și vom adăuga un număr mare de clase, librării, fișiere externe etc. Un program de version control are responsabilitatea de:

1. Stoca toate schimbările chiar dacă acestea sunt făcute de două persoane diferite
2. Documenta toate schimbările pentru a permite programatorilor să revizuiască schimbările.
3. Permite controlul asupra schimbărilor și returnarea la o versiune precedentă a programului în cazul că este necesar.

În esență acestea sunt posibilitățile de bază a GIT-ului, dar GitHub-ul este, precum reiese din denumire o platformă centrală care permite utilizatorilor să încarce proiecte(care pot fi private sau publice) care vor fi disponibile pentru alte persoane.

În prezent, marea majoritate a proiectelor open-source sunt stocate pe GitHub și permit doritorilor să contribuie prin adăugarea feat-urilor(cu aprobare ulterioară evident).

Avantajele folosirii GitHub-ului nu se termină aici. Un alt aspect important este faptul că acesta reprezintă și o rețea de socializare între programatori, care permite crearea echipelor, adresarea întrebărilor și răspunsurilor și alte aspecte sociale importante.

**Branching-ul în GIT**

Un concept important în GIT este cel de ramuri. Când un proiect este creat acesta are doar o ramură de bază (**master** branch), care se consideră versiunea de bază a programului vostru.

Există posibilitatea de a crea ramuri paralele, de obicei acest lucru se face când echipa testează o modificare fără să afecteze lucrul de bază a programului, puteți crea un număr oarecare de ramuri.

În final dacă aceste ramuri nu aduc rezultat pozitiv, ele pot fi pur și simplu șterse, sau dacă se decide că ele vor fi implementate în versiunea de bază a programului se execută un merge(unirea ramurii cu cea centrală).

Probleme pentru rezolvare.

### **Program 1:**

1. Creați repozitoriul vostru pe GitHub în care veți stoca examenul.
2. Faceți primul commit și push.

#### Continuare în logging. Trimiterea log-urilor.

După cum am discutat la lecția precedentă logging-ul oferă un număr mare de avantaje, dar pentru a primi mesajele de log de la utilizatori trebuie să le expediem cumva.

Pentru a realiza acest scop am ales opțiunea de a le trimite prin intermediul e-mailului imediat după ce acestea vor fi generate.

|  |
| --- |
| using System;  using System.Diagnostics;  using System.IO;  using System.Net;  using System.Net.Mail;  namespace LectiiAditionale  {  class Program  {  static int z = 5;  static int y = 0;  static void Main()  {  Program prog = new Program();  prog.Operatie();  }  private void Operatie()  {  try  {  int x = z / y;  Console.WriteLine(x);  File.WriteAllText("log.txt", "Executat cu succes.");  }  catch (Exception e)  {  //Numele metodei  var st = new StackTrace();  var sf = st.GetFrame(1);  //numele clasei //numele metodei //sistemul de operare  File.WriteAllText("log.txt", String.Format("Exceptia: {0} \nNumele clasei:{1} \nNumele metodei:{2} \nSistemul de operare:{3}", e.Message, GetType().Name, sf.GetMethod().Name, Environment.OSVersion.ToString()));  //Trimite mail  using (SmtpClient smtpClient = new SmtpClient())  {  var basicCredential = new NetworkCredential("nick@pandemonium-games.com", "Organic27402-");  using (MailMessage message = new MailMessage())  {  MailAddress fromAddress = new MailAddress("nick@pandemonium-games.com");  smtpClient.Host = "smtp.zoho.eu";  smtpClient.UseDefaultCredentials = false;  smtpClient.Credentials = basicCredential;  message.From = fromAddress;  message.Subject = "your subject";  message.IsBodyHtml = true;  message.Body = "<h1>your message body</h1>";  message.To.Add("nick.bota23@gmail.com");  message.Attachments.Add(new Attachment("log.txt"));  try  {  smtpClient.Send(message);  }  catch (Exception ex)  {  Console.Write(ex.Message);  }  }  }  }  }  }  } |

#### 

#### Continuare în WPF.

Scopuri:

1. Ordonare codului pentru ca fiecare clasă să respecte principiul Single Responsibility.
2. Îmbunătățirea UI-ului
3. Salvarea setărilor la închiderea programului
4. Încărcarea proiectului pe GitHub pentru acces comun.