### Modulul 8. Moștenirea în C#.

#### Moștenirea în C#

**Moștenirea** este unul din pilonii programării orientate pe obiect și unu din avantajele de bază. Moștenirea permite programatorilor să definească o clasă de bază de la care alte clase pot prelua funcționalitate. Acest lucru permite crearea unei logici generale într-o anumită clasă care va fi aplicată în mai multe situații, moștenirea acestei logici de către o clasă derivată și adaptarea specifică a acesteia la situația concretă.

Când discutăm de moștenire putem deosebi 2 tipuri de clase:

|  |  |
| --- | --- |
| 1. Clase de bază 2. Clase derivate |  |

Sintaxa declarării unei clase derivate în C#:

|  |
| --- |
| class ClasaDerivate : ClasaDeBaza  {  } |

**Esența moștenirii:**

Vom folosi moștenirea atunci avem nevoie să abstractizăm logica programului nostru în clase de bază care vor conține logică generală și clase derivate care vor face acțiuni mai specifice.

**Exemplu:**

|  |
| --- |
| Avem o clasă de bază **Vehicul,** care are metode generale precum Pornire(), Oprire() și variabile precum masa, viteza etc.  De la aceasta clasă vom deriva câteva clase: Mașină, Avion, Corabie etc.  Fiecare clasă va avea acces la metodele și variabilele declarate în clasa de bază Vehicul, deci va putea porni, opri și va avea viteza, masă etc.  Dar în același timp clasele derivate vor putea adăuga metode și variabile distincte de cele din clasă de bază.  Clasa Avion va avea variabila altitudine și metodele adiționale Decolare(), Aterizare().  Clasa Corabie va avea metodele RidicaPinze(), CoboaraPinze() etc. |

**Tipurile de moștenire în C#:**

1. **Moștenire simplă.** În cazul dat clasa derivată moștenește de la o singură clasă de bază și nu are la rândul ei alte clase derivate. Acest tip este folosit când avem nevoie de un mecanism simplu de abstractizare a datelor.



1. **Moștenire de mai multe nivele.** În această situație clasa derivată 1, are la rândul ei o altă clasă derivată care moștenește de la ea. Ambele posedă aceleași proprietăți ca și clasa de bază. Acest tip de moștenire se folosește când avem nevoie de un nivel mai mare de abstractizare.



1. **Moștenire în ierarhie.** În acest model de moștenirea avem o clasă de bază care servește drept model pentru mai multe clase derivate.



1. **Moștenire multiplă.** Deși moștenirea multiplă nu este posibilă în mod direct în C#, aceasta poate fi implementată cu ajutorul interfețelor.



1. **Moștenire hibridă.** Reprezintă o combinație între două sau mai multe tipuri de moștenire prezentate mai sus. **În exemplu moștenire în ierarhie + moștenire multiplă.**



**Exemplu:**

|  |
| --- |
| namespace IntroducereMostenire  {  class MarcaAuto  {  protected string marcaNume;  public void Claxon()  {  Beep(400, 1000);  Thread.Sleep(200);  Beep(400, 1000);  }  }  class ModelAuto: MarcaAuto  {  private string modelNume;  public string Model  {  get { return modelNume; }  }  public string Marca  {  get { return marcaNume; }  }  public ModelAuto(string \_marca, string \_model)  {  marcaNume = \_marca;  modelNume = \_model;  }  }  class Program  {  static void Main()  {  ModelAuto myCar = new ModelAuto("Audi", "A6");  WriteLine(myCar.Marca + " " + myCar.Model);  myCar.Claxon();  }  }  } |

Probleme pentru rezolvare.

### **Program 1:**

### Scrieți un program care veți crea o clasă de bază **User** cu variabilele **ID**, **password** și metoda **DateUser()**, care va printa “ID-ul: XXX are parola: YYY”.

### Creați o clasă derivată **UtilizatorInregistrat**.

### În interiorul ei creați un constructor cu 3 parametri string(ID, password și username).

1. Creați o variabilă locală **username.**
2. Creați o metodă locală **Logare()** cu sintaxa:

public void Logare()

{

Console.Write(“Utilizatorul ”+username+” logat cu succes”);

DateUser();

}

### Creați o clasă **Execuție(simplă, fără moștenire)**.

### În interiorul ei creați 3 utilizatori cu ajutorul constructorului cu parametri.

1. Apelați metoda logare la fiecare utilizator.

Rezolvare:

|  |
| --- |
| class User  {  protected string id;  protected string password;  protected void DateUser()  {  WriteLine("ID-ul:{0} are parola:{1}\n", id, password);  }  }  class UtilizatorInregistrat : User  {  private string username;  public UtilizatorInregistrat(string \_id, string \_password, string \_username)  {  id = \_id;  password = \_password;  username = \_username;  }  public void Logare()  {  WriteLine("Utilizatorul {0} logat cu succes.", username);  DateUser();  }  }  class Executie  {  private static void Main()  {  UtilizatorInregistrat user1 = new UtilizatorInregistrat("1292", "qwerty", "user123"); user1.Logare();  UtilizatorInregistrat user2 = new UtilizatorInregistrat("64203", "asdf", "etcam"); user2.Logare();  UtilizatorInregistrat user3 = new UtilizatorInregistrat("043", "zxcv", "user123"); user3.Logare();  }  } |

#### Metode virtuale și cuvântul cheie based

Cuvântul cheie virtual în C# definește o metodă/proprietate/indexator care poate fi moștenit, și **în același timp poate fi modificat în clasele derivate**.

Pentru a crea o metodă virtuală trebuie să creăm o metodă cu cuvântul cheie **virtual** în clasa de bază.

Apoi, în clasa derivată vom crea o metodă cu același nume cu cuvântul cheie **override.**

În cazul dat logica din metoda override va fi executată, dar cea din metoda virtuală nu.

**Exemplu:**

|  |
| --- |
| namespace Virtual  {  class ClasaBaza  {  protected virtual void Print()  {  WriteLine("1");  }  }  class ClasaDerivata : ClasaBaza  {  protected override void Print()  {  WriteLine("Print");  }  private static void Main()  {  ClasaDerivata clas = new ClasaDerivata();  clas.Print();  }  }  } |

Dar această funcționalitate nu este foarte utilă pentru că cu același succes am putea crea o altă funcție cu alt nume.

Utilitatea reală a metodelor virtuale devine evidentă atunci când le combinăm cu cuvântul cheie **base**.

Base se folosește pentru a crea o referință rapidă la clasa de bază în C#. Acesta va genera automat o referință la obiectul părinte, și în acest fel ne va permite să îl accesăm.

În combinație cu o metodă override cuvântul cheie base permite să executăm funcționalitatea din metoda virtuală **împreună** cu cea din metoda override. Ceea ce ne permite, în esență să creăm extensii ale metodelor din clasa de bază fără a rescrie funcționalitatea deja inclusă în ele.

**Exemplu:**

|  |
| --- |
| namespace Base  {  public class Utilizator  {  public string nume = "Vasile Qwerty";  public string locatie = "Cojusna";  public int varsta = 15;  public virtual void Informatii()  {  WriteLine("Nume: {0}", nume);  WriteLine("Locatie: {0}", locatie);  }  }  public class Detalii : Utilizator  {  public override void Informatii()  {  base.Informatii();  WriteLine("Virsta: {0}", varsta);  }  }  class Program  {  static void Main()  {  Detalii detalii1 = new Detalii();  detalii1.Informatii();  }  }  } |

Probleme pentru rezolvare.

### **Program 1:**

### Scrieți un program care veți crea o clasă de bază **Imobil** cu variabilele **suprafata**, **pret, nrCamere** și metoda virtuală **DateImobil()**, care va afișa “Imobilul dat are suprafata: X, Y camere și prețul Z”.

### Creați o clasă derivată **Apartament** cu variabilele **adresa**, **vinzator, numar telefon.** În interiorul clasei declarați o metodă override **DateImobil()** care va executa metoda din clasa de bază, apoi va afișa variabilele locale(adresa, vânzător, număr tel.)

### Adăugați un constructor cu 6 parametri în clasa Apartament.

### Creați o clasă independentă Execuție, în interiorul ei creați câteva apartamente cu ajutorul constructorului și apelați metoda DateImobil() din ele.

#### Folosirea cuvântului cheie sealed

Când discutăm despre moștenire în C# unul dintre cuvintele cheie importante este **sealed.**

Sealed delimitează un obiect/metodă care **NU poate fi moștenită.** Acesta este un mecanism util de protecție împotriva unor schimbări neprevăzute(ca și incapsularea discutată la lecția precedentă) și oferă un nivel adițional de control programatorilor asupra codului scris de ei.

În cazul în care încercăm să moștenim de la o clasă sealed, vom obține o eroare în compilator:

|  |  |
| --- | --- |
|  |  |

Aceasta este prima metodă de utilizare a cuvântului cheie sealed, a doua este de a crea metode care nu pot fi moștenite.

**Exemplu:**

|  |
| --- |
| namespace Sealed  {  class ClasaBaza  {  sealed protected virtual void Print()  {  WriteLine("Print");  }  }  class ClasaDerivata : ClasaBaza  {  protected override void Print()  {  WriteLine("Print");  }  }  } |

În exemplul de mai sus metoda Print este declarată virtuală în clasa de bază ceea ce semnifică că în clasa derivată putem să o moștenim și să schimbăm funcționalitatea ei, DAR din motivul că am indicat cuvântul cheie **sealed** când am creat metoda, moștenirea nu mai este posibilă.

Această tehnică este utilă atunci când vreți ca o metodă din clasa de bază să fie accesibilă, dar să nu poată fi moștenită și modificată.

#### Clase abstracte

În C# abstracția este o metodă de ascunde detaliile interne, arătând doar funcționalitatea.

Clasele abstracte nu pot fi folosite ca obiecte independente fiind doar o **bază** pentru alte clase care vor moșteni de la clasa abstractă.

**Clasele abstracte trebuie să conțină cel puțin o metodă abstractă și vor fi declarate cu ajutorul cuvântului cheie abstract înaintea class.**

**Exemplu:**

|  |
| --- |
| namespace Abstract  {  abstract class FiguraGeometrica  {  abstract public double Aria();  }  class Patrat : FiguraGeometrica  {  int lungimeLatura;  public Patrat(int \_latura) { lungimeLatura = \_latura; }  public override double Aria() { return lungimeLatura \* lungimeLatura; }  }  class Triunghi : FiguraGeometrica  {  int lungimeLatura;  public Triunghi(int \_latura)  {  lungimeLatura = \_latura;  }  public override double Aria()  {  return (Math.Sqrt(3) / 4 \* Math.Pow(lungimeLatura, 2));  }  }  class Executie  {  public static void Main()  {  Patrat p1 = new Patrat(6);  WriteLine("Aria patratului 1 = " + p1.Aria());  Patrat p2 = new Patrat(6);  WriteLine("\nAria patratului 2 = " + p1.Aria());  Triunghi t1 = new Triunghi(6);  WriteLine("\nAria triunghiului = " + t1.Aria());  }  }  } |

În exemplul de mai sus am utilizat clasa abstractă FiguraGeometrică care are metoda abstractă Aria în interior.

După cum observați o metodă abstractă nu trebuie să conțină logică, aceasta servește ca bază pentru funcțiile override din clasele derivate.

Din clasa abstractă am creat două clase derivate Patrat și Triunghi, ambele cu o formulă diferită pentru a calcula aria.

În rezultat, în clasa Executie am creat 2 obiecte de tip Patrat și un Triunghi și am apelat metoda Aria() care inițial era abstactă dar din motiv că a fost rescrisă în clasele derivate am primit răspunsul corect.

#### Analiza clasei de bază Object

Clasa de bază Object în C# servește drept clasă fundamentală pentru oricare altă clasă, indiferent dacă aceasta e creată de utilizator sau deja prezentă în Framework. Clasa Object face parte din librăria System.

Dacă transmitem Object ca parametru al unei metode acea metodă va primi orice obiect de orice clasă ceea ce o face un instrument flexibil de generalizare.

În același timp putem folosi Object în combinație cu reflecția pentru a returna tipul clasei primite, numele obiectului etc.

**Exemplu:**

|  |
| --- |
| namespace ObjectBase  {  class ClasaBaza  {}  class ClasaDerivata1 : ClasaBaza  {}  class ClasaDerivata2 : ClasaBaza  {}  class Executie  {  private static void Main()  {  ClasaBaza clasaBaza = new ClasaBaza();  ClasaDerivata1 clasa1 = new ClasaDerivata1();  ClasaDerivata2 clasa2 = new ClasaDerivata2();  WriteLine(ComparaObiecte(clasa1, clasa1));  WriteLine(ComparaObiecte(clasa1, clasa2));  WriteLine(ComparaObiecte(clasa1, clasaBaza));  }  private static bool ComparaObiecte(object \_ob1, object \_ob2)  {  return \_ob1.Equals(\_ob2);  }  }  } |

În exemplul de mai sus am creat o metodă de tip bool care compară două obiecte și returnează dacă acestea sunt egale între ele. Din motiv că C# este un limbaj bazat pe referințe, metoda va returna true doar atunci când referințele primite vor indica spre același obiect, dacă obiectele sunt diferite vom primi fals, chiar dacă ele sunt din aceeși clasă.

#### Împachetare, despachetare(boxing, unboxing)

Împachetarea și despachetarea sau boxing și unboxing sunt operațiile care permit convertirea dinamică a valorilor în obiecte și viceversa

**Exemplu:**

|  |
| --- |
| namespace BoxingUnboxing  {  class Boxing  {  static public void Main()  {  int num = 2020;  object obj = num;  WriteLine("Valorea numarului este:{0} si tipul este:{1}", num, num.GetType());  WriteLine("Valorea objectului este:{0} si tipul este:{1}", obj, obj.GetType());  WriteLine("Valorea sumei este:{0}", (int)obj + num);  }  }  } |

Conceptele date nu vor avea o valoarea practică majoră dar vă vor ajuta să sesizați că C# este capabil de a face operații între obiect și variabile primitive prin intermediul conceptului dat.

### Modulul adițional 1. Continuare în Async. Requesturi spre servere și prelucrarea răspunsurilor.

La lecția precedentă am discutat despre operațiile Async și din ce motiv trebuie să așteptăm răspunsul unor metode înainte de a începe altele, acest lucru devine foarte evident atunci când încercăm să preluăm informații de pe server și nu știm când îl vom primi(sau dacă îl vom primi).

**Exemplu:**

|  |
| --- |
| namespace AsyncRequest  {  using System;  using System.Net;  using System.IO;  using System.Text;  class Request  {  public static StreamReader serverResponse;  static void Main()  {  string urlAddress = "http://google.com";  SendRequest(urlAddress);  PrintData(serverResponse);  }  public static void SendRequest(string \_url)  {  HttpWebRequest request = (HttpWebRequest)WebRequest.Create(\_url);  HttpWebResponse response = (HttpWebResponse)request.GetResponse();  Stream receiveStream = response.GetResponseStream();  StreamReader readStream = null;  readStream = new StreamReader(receiveStream, Encoding.GetEncoding(response.CharacterSet));  serverResponse = readStream;  }  public static void PrintData(StreamReader \_data)  {  string raspuns = \_data.ReadToEnd();  Write(\_data);  }  }  } |

În exemplul de mai sus am scris un program care face request către un anumit URL și ar trebui să returneze tot codul HTML în formă de text.

Din motiv că nu a fost implementat mecanismul de așteptare variabila noastră rămâne goală.

Pentru a face programul nostru să funcționeze corect trebuie să implementăm operațiile Async.

**Exemplu:**

|  |
| --- |
| namespace AsyncRequest  {  using System.Net;  using System.IO;  using System.Text;  using System.Threading.Tasks;  class Request  {  public static StreamReader serverResponse;  public static string urlAddress = "http://interflix.me";  static void Main()  {  MetodaGeneral();  ReadKey();  }  public static async Task MetodaGeneral()  {  await SendRequest(urlAddress);  await PrintData(serverResponse);  }  public 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, Encoding.GetEncoding(response.CharacterSet));  serverResponse = readStream;  });  }  public static async Task PrintData(StreamReader \_data)  {  await Task.Run(() =>  {  string raspuns = \_data.ReadToEnd();  Write(raspuns);  });  }  }  } |

Mai sus avem deja varianta corectă a programului. Pentru a demonstra am găsit cel mai lent website de pe internet și am făcut un request către acesta. După cum vedeți programul trebuie să aștepte câteva secunde chiar și fără a arăta interfața grafică, dar din motiv că am folosit operația de așteptare codul HTML a fost afișat cu succes.

### Modulul adițional 2. Finisarea playerului audio în WPF și crearea aplicației Windows.

### Probleme.

### **Program 1:**

### Scrieți un program care veți crea o clasă de bază **Suprafata** cu variabilele **lungime si latime** și 2 metode: **Aria**(care va returna aria suprafetei lungime\*latime), **DetaliiSuprafata(**care va afisa la consola lungimea, latimea si aria).

### Creați o clasă derivată **Comanda** care derivă de la Suprafata

### Adăugați o variabilă privată pretPerMetru.

### Creați un constructor cu 3 parametri float(lungime, latime și pret).

1. Creați o metodă locală **DetaliiFinale()** care va chema metoda **DetaliiSuprafata** din clasa de bază și va afișa prețul final al comenzii.

### Creați o clasă **Execuție(simplă, fără moștenire)**. Creați unul sau mai multe obiecte de tip Comanda, folosiți constructorul și afișați detaliile despre comanda data.

### 

### **Program 2:**

### Scrieți un program care veți crea o clasă de bază **Calcul** cu variabilele **x**, **y, z** și metoda **virtuală** de tip float **Rezultat()**, care va returna rădăcina pătrată a sumei acestor numere.

### Creați un constructor cu 3 parametri în interiorul clasei date.

### Creați o clasă derivată cu numele **CalculSecundar** cu o variabilă locală **q** și folosiți o metodă **override** Rezultat() pentru a aduna **q3** la rezultatul din clasa de bază. Adăugați un constructor cu 4 parametri în clasa dată.

### Creați o clasă independentă Execuție, în interiorul ei creați un obiect de tip Calcul, și alt obiect de tip CalculSecundar. Apelați metoda Rezultat() din ambele, afișați-l, apoi adunați-le între ele.

### **Program 3:**

### Similar cu programul exemplu de mai sus, creați un program cu clasa abstractă **CorpuriGeometrice** care va include metoda abstractă **Volum**.

### Creați două clase derivate Cub și Paralelipiped, cu constructori și variabile specifice și metode override Volum.

### Utilizați constructorii și afișați volumul corpurilor geometrice.

Volum cub = latura3 Volum paralelipiped = latura1\*latura2\*latura3

### **Program 4:**

### Scrieți un program care veți crea o clasă de bază **Calcul** cu o **metodă abstractă de tip object** rezultat.

### Creați câteva clase derivate de tip calcul(Adunare, Scădere, Produs etc.) care vor avea constructori și variabile private și folosiți metode **override** pentru a returna rezultatul operației între variabile.

### Creați câte un obiect de fiecare tip de operație cu ajutorul constructorului și afișați rezultatul.

### **Program 5:**

### Scrieți un program care veți avea 2 metode. Prima va afișa 50 de numere random cu un interval de 0,1 secunde între ele. A doua va afișa 10 numere random cu un interval de 0,5 secunde între ele.

1. Folosiți operații Async și comanda Wait pentru a le executa în ordine.

### **Opțional**

### **Program 6:**

### Scrieți un program care va transmite requesturi la mai multe website-uri și va returna timpul în care a fost primit un răspuns.

### Trimiteți 10 requesturi la diferite site-uri și grupați-le după cât de rapid au răspuns.

Exemplu:

|  |
| --- |
| namespace AsyncRequest  {  using System.Net;  using System.IO;  using System.Text;  using System.Threading.Tasks;  using System.Diagnostics;  class Request  {  public static StreamReader serverResponse;  public static string urlAddress = "http://interflix.me";  private static TimeSpan requestTime;  static void Main()  {  var t = Task.Run(() => SendRequest(urlAddress));  t.Wait();  var z = Task.Run(() => PrintData(serverResponse));  z.Wait();  }  public static async Task SendRequest(string \_url)  {  await Task.Run(() =>  {  Stopwatch timer = new Stopwatch();  timer.Start();  HttpWebRequest request = (HttpWebRequest)WebRequest.Create(\_url);  HttpWebResponse response = (HttpWebResponse)request.GetResponse();  Stream receiveStream = response.GetResponseStream();  StreamReader readStream = null;  readStream = new StreamReader(receiveStream);  serverResponse = readStream;  timer.Stop();  requestTime = new TimeSpan(0, 0, 0, (int)(timer.ElapsedMilliseconds / 1000));  });  }  public static async Task PrintData(StreamReader \_data)  {  await Task.Run(() =>  {  string raspuns = \_data.ReadToEnd();  Write(raspuns);  WriteLine("\nRequest executat in {0}.", requestTime);  });  }  }  } |