### Modulul 9. Interfețe în C#.

#### Ce este o interfață?

**O interfață** este similară cu o clasă abstractă prin ceea că aceasta conține un grup de metode, proprietăți sau indexatori abstracți pe care clasele derivate vor trebui să le implementeze.

Dar există câteva diferențe importante dintre o interfață și o clasă abstractă:

1. O interfață nu poate conține variabile și nu poate defini metodele, proprietațile etc.
2. O structură poate moșteni de la o interfață, dar nu de la o clasă abstractă.
3. Toți membrii unei interfețe sunt by default abstract și public.
4. Interfața nu poate conține un constructor.
5. O clasă sau structură poate moșteni de la multiple interfețe, ceea ce semnifică că interfețele sunt unica metodă de implementare a moștenirii multiple în C#.

**În esență** o interfață reprezintă un șablon obligatoriu pe care clasele derivate trebuie să-l respecte, fără excepții.

#### Sintaxa creării și utilizării interfețelor.

În C# o interfață se declară cu ajutorul cuvântului cheie **interface** urmat de numele ales al interfeței.

Este o practică bună de începe numele unei interfețe cu I pentru a face clar imediat că aceasta este o interfață, cu clasă sau structură.

În interiorul interfeței programatorul decide care metode, indexatori, proprietăți vrea să le includă. Toate aceste elemente nu au o implementare locală, din motiv că sunt abstracte, din acest motiv ele vor trebui definite în fiecare clasă sau structură care va implementa interfața dată.

**Sintaxa:**

|  |
| --- |
| interface IProgram  {  string Denumire { get; set; }  bool Deschide();  bool Inchide();  void Executa(string \_cod);  } |

Atrageți atenția la câteva detalii în sintaxa de sus:

1. Nici una dintre metode sau variabile nu are nivel de acces(private, public etc.), pentru că este public abstract by default.
2. Metoda Executa nu are implementare, pentru că ea reprezintă doar un șablon.

#### Exemple de utilizare a interfețelor

După cum am menționat mai devreme utilizarea de bază a unei interfețe este de a crea un șablon pentru una sau mai multe clase.

**Exemplu:**

|  |
| --- |
| namespace Exemple  {  interface IDispozitiv  {  string Tip { get; set; }  void Pornire();  void Oprire();  void Restart();  void Statut();  }  class Calculator: IDispozitiv  {  public string Tip { get; set;}  private bool pornit;  public void Pornire()  {  ForegroundColor = ConsoleColor.Green;  WriteLine("\nPornire dispozitiv...");  Tip = this.GetType()+"";  pornit = true;  }  public void Oprire()  {  ForegroundColor = ConsoleColor.Red;  WriteLine("\nOprire dispozitiv...");  pornit = false;  }  public void Restart()  {  ForegroundColor = ConsoleColor.Yellow;  WriteLine("\n{0} a fost restartat.", Tip);  }  public void Statut()  {  WriteLine("{0} este pornit:{1}", Tip, pornit);  }  private static void Main()  {  Calculator calc = new Calculator();  calc.Pornire(); calc.Statut();  calc.Oprire(); calc.Statut();  calc.Restart();  ReadKey();  }  }  } |

În exemplul de mai sus vedem cum poate fi utilizată o interfață pentru a crea un set de metode și variabile care trebuie necesar implementate în clasa derivată Calculator. Definirea funcționalității se efectuază în clasa Calculator, iar interfața IDispozitiv servește doar ca un “contract” care obligă clasa derivată să implementeze un set de metode și variabile.

Probleme pentru rezolvare.

### **Program 1:**

### Creați o interfață IAnimal, care va include cel puțin o variabilă și o metodă.

1. Implementați această interfață în 3 clase derivate, fiecare clasă având o implementare diferită.
2. Afișați rezultatul.

Rezolvare:

|  |
| --- |
| interface IAnimal  {  string Specie { get; set; }  void Detalii();  }  class Tigru : IAnimal  {  public string Specie { get; set; }  public void Detalii()  {  ForegroundColor = ConsoleColor.DarkYellow;  Specie = this.GetType().Name + "";  WriteLine("\n" + Specie);  Beep(300, 1000);  }  }  class Iepure : IAnimal  {  public string Specie { get; set; }  public void Detalii()  {  ForegroundColor = ConsoleColor.White;  Specie = this.GetType().Name + "";  WriteLine("\n" + Specie);  Beep(1000, 300);  }  }  class Girafa : IAnimal  {  public string Specie { get; set; }  public void Detalii()  {  ForegroundColor = ConsoleColor.Yellow;  Specie = this.GetType().Name + "";  WriteLine("\n" + Specie);  Beep(500, 500);  }  }  class Executie  {  private static void Main()  {  Tigru tigru = new Tigru(); tigru.Detalii();  Iepure iepure = new Iepure(); iepure.Detalii();  Girafa girafa = new Girafa(); girafa.Detalii();  ReadKey();  }  } |

#### Folosirea interfețelor multiple

Unul din avantajele principale ale interfețelor în C# este faptul că acestea permit unei clase să moștenească de la mai multe interfețe, ceea ce face posibilă moștenirea multiplă.

De exemplu în cazul în care aveți nevoie să creați 2 interfețe separate cu funcționalitate diferită: tranzacție și procurare obiect.

**Exemplu:**

|  |
| --- |
| interface ITranzactie  {  float SumaTranzactiei { get; set; }  string IDTranzactie { get; set; }  void DetaliiTranzactie();  }  interface IActiv  {  float ValoareaActivului { get; set; }  string DenumireActiv { get; set; }  void DetaliiActiv();  }  class ProcurareActiv : ITranzactie, IActiv  {  //Tranzactie  public float SumaTranzactiei { get; set; }  public string IDTranzactie { get; set; }  public void DetaliiTranzactie() { WriteLine("Tranzactia:{0} in suma de {1} lei.", IDTranzactie, SumaTranzactiei); }  //Activ  public float ValoareaActivului { get; set; }  public string DenumireActiv { get; set; }  public void DetaliiActiv() { WriteLine("Activul:{0} are valoarea de {1} lei.", DenumireActiv, ValoareaActivului); }  //Constructor  public ProcurareActiv(float \_sumaTranz, string \_id, string \_denumireActiv, float \_valoareActiv)  {  SumaTranzactiei = \_sumaTranz;  IDTranzactie = \_id;  DenumireActiv = \_denumireActiv;  ValoareaActivului = \_valoareActiv;  }  //Procurare  private DateTime azi = DateTime.Now;  public void DetaliiProcurare() { WriteLine("Pe data de {0}, a fost procurat activul {1} cu suma de {2} lei. Tranzactia are ID-ul: {3}", azi, DenumireActiv, SumaTranzactiei, IDTranzactie); }  //Executie  private static void Main()  {  ProcurareActiv proc1 = new ProcurareActiv(15000, "ABE154", "Printer XYZ", 17000);  proc1.DetaliiProcurare();  }  } |

În exemplul de mai sus vedem cum pot fi combinate 2 interfețe diferite și implementate în aceeași clasă. Interfața ITranzacție poate fi folosită pentru a descrie orice tranzactie din companie, și interfața IActiv poate fi folosită pentru a descrie orice activ, dar combinându-le am creat o clasă intermediară ProcurareActiv, care o putem folosi de fiecare dată când compania va procura un utilaj.



#### Referințe la interfață

Un alt avantaj important al interfețelor în C# este că acestea permit crearea referințelor la ele, ceea ce ne permite să accesăm anume metodele și variabilele de care avem nevoie.

Acest lucru devine deosebit de util atunci când avem o clasă care implementează mai multe interfețe și numărul de proprietăți și metode crește substanțial.

**Sintaxa:**

|  |
| --- |
| interface IProgram  {  string Denumire { get; set; }  }  interface IUtilizator  {  string Nume{ get; set; }  }  class UtilizatorProgram: IProgram, IUtilizator  {  public string Denumire { get; set; }  public string Nume { get; set; }  //Constructor  public UtilizatorProgram(string \_denumireProgram, string \_numeUtilizator)  {  Denumire = \_denumireProgram;  Nume = \_numeUtilizator;  }  private static void Main()  {  **//Initializam obiectul de tipul clasei**  UtilizatorProgram util1= new ProcurareActiv(“Visual Studio”, "Ion");  **//Cream referinta la interfata IProgram**  IProgram refProgram;  refProgram= util1;  Console.WriteLine(refProgram.Denumire);  **//Cream referinta la interfața IUtilizator**  IUtilizator refUtilizator;  refUtilizator= proc1;  Console.WriteLine(refUtilizator.Nume);  }  } |

**Exemplu:**

|  |
| --- |
| interface ITranzactie  {  float SumaTranzactiei { get; set; }  string IDTranzactie { get; set; }  void DetaliiTranzactie();  }  interface IActiv  {  float ValoareaActivului { get; set; }  string DenumireActiv { get; set; }  void DetaliiActiv();  }  class ProcurareActiv : ITranzactie, IActiv  {  //Tranzactie  public float SumaTranzactiei { get; set; }  public string IDTranzactie { get; set; }  public void DetaliiTranzactie() { WriteLine("\nTranzactia:{0} in suma de {1} lei.", IDTranzactie, SumaTranzactiei); }  //Activ  public float ValoareaActivului { get; set; }  public string DenumireActiv { get; set; }  public void DetaliiActiv() { WriteLine("\nActivul:{0} are valoarea de {1} lei.", DenumireActiv, ValoareaActivului); }  //Constructor  public ProcurareActiv(float \_sumaTranz, string \_id, string \_denumireActiv, float \_valoareActiv)  {  SumaTranzactiei = \_sumaTranz;  IDTranzactie = \_id;  DenumireActiv = \_denumireActiv;  ValoareaActivului = \_valoareActiv;  }  //Procurare  private DateTime azi = DateTime.Now;  public void DetaliiProcurare() { WriteLine("Pe data de {0}, a fost procurat activul {1} cu suma de {2} lei. Tranzactia are ID-ul: {3}", azi, DenumireActiv, SumaTranzactiei, IDTranzactie); }  //Executie  private static void Main()  {  ProcurareActiv proc1 = new ProcurareActiv(15000, "ABE154", "Printer XYZ", 17000);  proc1.DetaliiProcurare();  //Cream referinta la interfete  ITranzactie refTranzactie;  refTranzactie = proc1;  refTranzactie.DetaliiTranzactie();  //Cream referinta la interfete  IActiv refActiv;  refActiv = proc1;  refActiv.DetaliiActiv();  //Folosim referintele pentru operatii  float diff = refTranzactie.SumaTranzactiei - refActiv.ValoareaActivului;  if (diff < 0) WriteLine("\nCompania a procurat activul cu {0} lei mai ieftin decit valoarea acestuia.", Math.Abs(diff));  if (diff > 0) WriteLine("\nCompania a procurat activul cu {0} lei mai scump decit valoarea acestuia.", Math.Abs(diff));  if (diff == 0) WriteLine("\nCompania a procurat activul la valoarea lui reala de {0} lei.", refActiv.ValoareaActivului);  }  } |

În exemplul dat am utilizat problema elaborată anterior, dar de data aceasta am creat două referințe la interfețele pe care le-am implementat. În continuare am folosit referințele pentru a identifica diferența între prețul achitat pentru utilaj și valoarea lui reală.

#### Cuvântul cheie is

Referințele la interfețe au și o utilitate adițională care devine accesibilă cu ajutorul cuvântului cheie is. **Cuvântul cheie is ne permite să vedem dacă o interfață a fost implementată într-un obiect.**

Dacă interfața este implementată vom primi true, dacă nu - vom primi false.

**Sintaxa:**

|  |
| --- |
| interface Interface1{}  interface Interface2{}  interface Interface3{}  class InterfaceImplement : Interface1, Interface2  {  private static void Main()  {  InterfaceImplement proc1 = new InterfaceImplement();  if (proc1 is Interface1) WriteLine("Obiectul implementeaza interfata 1.");  if (proc1 is Interface2) WriteLine("Obiectul implementeaza interfata 2.");  if (proc1 is Interface3) WriteLine("Obiectul implementeaza interfata 3.");  }  } |

**Exemplu:**

|  |
| --- |
| interface IButton { void Click(); }  interface IImagine { float Dimensiune(); }  interface ISpatiuGol { void Click(); }  class Buton : IButton  {  public void Click() { WriteLine("Butonul a fost apasat."); }  }  class Imagine : IImagine  {  public float Dimensiune() { return 10; }  public void Click() { WriteLine("Nu se intimpla nimic."); }  }  class ImagineClickable : IImagine, IButton  {  public float Dimensiune() { return 10; }  public void Click() { WriteLine("Imaginea a fost apasata."); }  }  class SpatiuGol : ISpatiuGol  {  public void Click() { WriteLine("Nu se intimpla nimic."); }  }  class Executie  {  private static void Main()  {  SpatiuGol buton1 = new SpatiuGol();  if (buton1 is IButton) { buton1.Click(); }  }  } |

În exemplul de mai sus am demonstrat cum poate fi folosit cuvântul cheie is în cazul în care avem mai multe interfețe. Pentru început am declarat 4 interfețe diferite, dar toate din ele conțin metoda Click().

Vom executa metoda doar dacă obiectul creat include interfața IButon, pentru în mod opus elementul nu poate fi apăsat.

După cum observați cuvântul cheie is servește drept un instrument pentru crearea unei verificări logice, similar cu metoda Contains care o folosim în cadrul unui tablou să verificăm dacă acesta conține un element X.

#### Proprietăți și indexatori ai interfețelor

Analogic cu metodele, indexatorii și proprietățile unei interfețe nu trebuie să conțină un corp, ci doar un get set simplu, fără logică adițională. Dar în clasele derivate din interfață puteți schimba această logică cum doriți.

**Exemplu:**

|  |
| --- |
| interface IUserInfo  {  string Name { get; set; }  string this[int index] { get; set; }  }  class UserInfo : IUserInfo  {  string myName;  public string Name  {  set { myName = value+"a"; }  get { return myName; }  }  public string this[int \_index]  {  set { myName = value; }  get { return myName[\_index].ToString(); }  }  }  class Program  {  static void Main()  {  UserInfo user1 = new UserInfo();  user1.Name = "Alan";  WriteLine(user1.Name);  WriteLine(user1[1]);  }  } |

#### Moștenirea interfețelor

Interfețele, ca și clasele pot moșteni de la alte interfețe. În cazul dat interfața secundară va avea acces la toate metodele, proprietățile și indexatorii interfeței de bază. Iar clasa derivată de la interfața secundară va trebui să implementeze toate cele enumerate.

Spre deosebire de clase în schimb i**nterfețele nu pot utiliza cuvântul cheie sealed pentru a interzice moștenirea**. De asemenea interfețele nu pot folosi cuvântul cheie abstract din motive evidente, ele deja sunt abstracte.

Când vorbim de moștenirea interfețelor nu vom utiliza cuvântul cheie **override** care a fost utilizat la moștenirea claselor, în schimb vom folosi cuvântul cheie new cu același scop.

**Sintaxa:**

|  |
| --- |
| interface InterfataBaza  {  void Exemplu();  }  interface InterfataDerivata: InterfataBaza  {  void Exemplu2();  }  class ClasaInterfataDerivata : InterfataDerivata  {  public void Exemplu() { }  public void Exemplu2() { }  } |

Spre deosebire de clase o interfață poate moșteni de la mai multe interfețe în același timp.

**Sintaxa:**

|  |
| --- |
| interface InterfataBaza  {  void Exemplu1();  }  interface InterfataBaza2  {  void Exemplu2();  }  interface InterfataDerivata : InterfataBaza, InterfataBaza2  {  void Exemplu3();  }  class ClasaInterfataDerivata : InterfataDerivata  {  public void Exemplu1() { }  public void Exemplu2() { }  public void Exemplu3() { }  } |

### Modulul adițional.

### 4 Principii de bază OOP

### Principiul DRY și KISS.

### Principiile SOLID

### Modelul MVC

#### 4 Principii de bază OOP

Una din întrebările cele mai frecvente pentru candidații la poziția de Junior Developer C#(și alte limbaje OOP) este de a explica cele 4 principii ale programării orientate pe obiect.

Ele sunt: Incapsularea, Abstractizarea, Moștenirea și Polimorfismul.

1. **Incapsularea** presupune că toate datele din interiorul unei clase trebuie să fie innacesibile pentru clasele din exterior. În cazul în care datele sunt necesare în alte clase folosiți proprietăți.

|  |  |
| --- | --- |
| **Greșit**  public class Student  {  public string FirstName;  public string LastName;  public int Age;  public string Id;  } | **Corect**  public class Student  {  private string firstName;  private string lastName;  private int age;  private string id;  public string FirstName  {  get{ return firstName;}  set{ firstName = value;}  }  public string LastName  {  get{ return lastName;}  set{ lastName = value;}  }  public int Age  {  get{ return age;}  set{ age= value;}  }  public string ID  {  get{ return id;}  set{ id= value;}  }  } |

1. **Moștenirea** presupune că toate clasele care au o logică de bază comună trebuie să moștenească de la o clasă de bază și să adăugați logica specifică în clasele derivate.

|  |  |
| --- | --- |
| **Greșit**  public class Masina  {  public float VitezaMaxima;  public float Masa;  public string Marca;  public void Pornire(){};  public void Oprire(){};  public void DeschidePortbagaj(){};  }  public class Camion  {  public float VitezaMaxima;  public float Masa;  public string Marca;  public void Pornire(){};  public void Oprire(){};  public void AtaseazaRemorca(){};  public void ScoateRemorca(){};  }  public class Bicicleta  {  public float VitezaMaxima;  public float Masa;  public string Marca;  public void Pornire(){};  public void Oprire(){};  public void RidicareIntroRoata(){};  } | **Corect**  public class Vehicol  {  protected float VitezaMaxima;  protected float Masa;  protected string Marca;  protected void Pornire(){};  protected void Oprire(){};  }  public class Masina: Vehicol  {  private void DeschidePortbagaj(){};  }  public class Camion: Vehicol  {  private void AtaseazaRemorca(){};  private void ScoateRemorca(){};  }  public class Bicicleta: Vehicol  {  private void RidicareIntroRoata(){};  } |

1. **Abstractizarea** reprezintă procesul de creare a unui șablon pe care clasele voastre îl vor implementa ulterior. Procesul de abstractizare asigură o structură rigidă care trebuie urmată de toate clasele de un anumit tip, ceea ce va ajuta să menține o structură comună chiar și pentru o echipă mare de programatori.

|  |  |
| --- | --- |
| **Greșit**  public class Triunghi  {  public float arie;  public float Perimetru;  public float ipotenuza;  public float Latura1;  }  public class Patrat  {  public float arie;  public float Perimetru;  public float Latura;  }  public class Cerc  {  public float arie;  public float Perimetru;  public float Raza;  }  ... | **Corect**  interface IFigura  {  float Arie{get; set;}  float Perimetru{get; set;}  float Raza{get; set;}  }  public class Triunghi  {  public float Arie(){...}  public float Perimetru(){...}  public float Raza(){...}  }  public class Patrat  {  public float Arie(){...}  public float Perimetru(){...}  public float Raza(){...}  }  public class Triunghi  {  public float Arie(){...}  public float Perimetru(){...}  public float Raza(){...}  }  public class Cerc  {  public float arie;  public float Perimetru;  public float Latura;  } |

1. **Polimorfismul** este abilitatea unui obiect de a avea mai multe forme. Dar în cazul POO se referă la posibilitatea de a supraîncărca metodele și operatorii. Principiul polimorfismului încurajează crearea unor metode care acceptă mai multe tipuri de date și le prelucrează corespunzător în loc să trebuiască să creați o metodă respectivă pentru orice operație.

|  |  |
| --- | --- |
| **Greșit**  public class Concatinare  {  private string cuv1=”a”;  private string cuv2=”b”;  private static void Main()  {  string cuv3 = ””;  for(int i=0; i<(cuv1.Length); i++)  {  cuv3[i] = cuv1[i];  }  for(int i=0; i<(cuv2.Length); i++)  {  cuv3[i+1] = cuv1[i];  }  }  } | **Corect**  public class Concatinare  {  private string cuv1=”a”;  private string cuv2=”b”;  private static void Main()  {  Console.Write(cuv1+cuv2);  }  } |

#### Principiul DRY și KISS

Alte principii important în POO sunt **DRY**(Don’t Repeat Yourself) și **KISS**(Keep It Simple Stupid).

1. **DRY** - presupune exact ceea ce conține numele, evitarea repetițiilor. Pentru a respecta acest principiu se recomandă împărțirea codului în librării care pot fi reutilizate în cât mai multe locuri(de exemplu o singură clasă care salvează datele, nu repetarea procesului în fiecare loc etc.)

De obicei putem vedea foarte rapid dacă principiul DRY se respectă într-un program doar din cantitatea de cod scrisă, cu cât multe rânduri de cod, cu atât mai mică probabilitatea că DRY a fost implementat.

1. **KISS** sau Keep It Simple Stupid - semnifică că codul pe care îl scrieți trebuie să fie cât mai simplu, cât mai scurt și cât mai clar.

**Exemplu**:

|  |
| --- |
| public String weekday1(int day)  {  switch (day) {  case 1:  return "Monday";  case 2:  return "Tuesday";  case 3:  return "Wednesday";  case 4:  return "Thursday";  case 5:  return "Friday";  case 6:  return "Saturday";  case 7:  return "Sunday";  default:  throw new InvalidOperationException("day must be in range 1 to 7");  }  }  public String weekday2(int day)  {  if ((day < 1) || (day > 7)) throw new InvalidOperationException("day must be in range 1 to 7");  string[] days =  {  "Monday",  "Tuesday",  "Wednesday",  "Thursday",  "Friday",  "Saturday",  "Sunday"  };  return days[day - 1];  } |

Cea mai simplă metodă de a verifica dacă principiul KISS este respectat este să arătați codul pe care l-ați scris unui programator care nu este familiar cu el. Dacă acesta are nevoie de mult timp pentru a înțelege ce face codul și cum este structurat atunci este nevoie de simplificare.

#### Principiile SOLID

Acronimul **S.O.L.I.D.** reprezintă un set de principii, fiecare literă având o semnificație specifică și anume:

* **Single-responsibility(responsabilitate unică)**. Fiecare clasă trebuie să îndeplinească o singură funcție.
* **Open-closed principle(deschis spre extindere închis spre modificare).** Principiul dat presupune că clasele care îndeplinesc o anumită funcție trebuie să fie extinse prin moștenire în cazul în care vrem să adăugăm funcționalități noi, dar acestea nu trebuie modificate la rădăcina lor.

<https://dotnettutorials.net/lesson/open-closed-principle/>

* **Liskov substition principle.** Presupune că un program trebuie să continue să funcționeze corect chiar și atunci când substituim un obiect dintr-o clasă cu alt obiect din subclasa acestuia. Într-o variantă mai simplă de explicație acest principiu presupune că clasele voastre de bază nu trebuie să conțină logică care contrazice logica claselor derivate. (De exemplu clasa de bază Calcul poate avea rezultat negativ, dar o clasă derivată Aria poate avea rezultat negativ).

<https://dotnettutorials.net/lesson/liskov-substitution-principle/>

* **Interface segregation.** Acest principiu este foarte simplu și presupune că nici un obiect care folosește o interfață nu trebuie să folosească o metodă, proprietate care nu o folosește. De exemplu avem o interfață angajat care include metodele Curatare(), Contabilitate() etc. și de la această interfață moștenesc toți angajații companiei, dar majoritatea angajaților nu au nevoie de metoda Curatare() sau Contabilitate(). Principiul de segregare a interfețelor presupune că este mai bine să creați interfețe separate și specializate în acest caz.

<https://dotnetcoretutorials.com/2019/10/27/solid-in-c-interface-segregation-principle/>

* **Dependency inversion.** Presupune că clasele pe care le creăm nu trebuie să depindă complet de referințele lor către alte clase. În cazul în care una din clasele referință lipsește sau nu funcționează corect clasa care funcționează trebuie să includă metode de evitare a erorilor/excepțiilor generate de dependețele date, sau să identifice metode de a evita aceste excepții complet.

<https://www.tutorialsteacher.com/ioc/dependency-inversion-principle>

Probleme pentru rezolvare.

### **Program 1:**

### Creați două interfețe: IMamifer și IPasare cu o proprietate și o metodă.

### Creați o clasă Pinguin care implementează ambele interfețe.

### Apelați metodele din fiecare interfață în clasă, în Main.

### Creați o metodă locală în clasă care va afișa proprietățile din fiecare interfață.

### **Program 2:**

### Folosiți problema creată în punctul precedent, dar de data aceasta creați două referințe la interfețele pe care le-ați implementat.

### Folosiți referințele pentru a accesa metodele din interfețe.

### Folosiți referințele pentru a efectua o operație cu proprietățile din interfețe.

### **Program 3:**

### Creați 3 interfețe cu o metodă și o variabilă(proprietate).

### Creați câteva obiecte care implementează una sau mai multe interfețe la alegere.

### Dacă obiectul implementează interfața 1 și 2 faceți o operație cu variabilele implementate din interfața 1 și 2.

### **Program 4:**

### Creați o interfață listă care va include un tablou de string-uri(proprietate) și un indexator.

### Într-o clasă derivată din această interfață declarați ca indexatorul să returneze cuvântul[index-2] dacă acest lucru e posibil, dacă indexul este mai mic ca 0 returnați ultimul cuvânt din tablou.