EIM&IDP-LABORATOR 3

Limbajul C#; Facilitati esentiale

În aceasta lucrare vom prezenta, pe scurt caracteristicile limbajului C# pe care un programator pe platforma .NET Core trebuie să le cunoască.

Colectii in limbajul C#. Clasele colecție sunt generice in C#. Cu mici excepții, toate clasele și interfețele colectie se găsesc în spațiul de nume System.Collections.Generic. Toate clasele colectie generice implementează interfața ICollection <T>, care conține funcționalitatea de bază împărtășită de toate colecțiile generice. Interfata ICollection <T> extinde interfata IEnumerable<T>. Clasa Enumerable<T> implementeaza interfata IEnumerable<T> si adauga o multime de metode de extensie (statice) (metodele LINQ)

Metode ale interfetei **ICollection<T>:**

- **Add(T)** - Adaugă o instanță de tipul T în colecție
- **Clear()** - Elimină toate articolele din colecție
- **Contains(T)** – Întoarce true dacă colecția conține o instanță dată de tipul T
- **CopyTo (T [], int)** - Copiază conținutul colecției într-un tablou, începând de la indicele specificat
- **Count** - Returnează numărul de articole din colecție
- **GetEnumerator()** - returnează un **IEnumerator<T>** pentru elementele colectiei
- **IsReadOnly** - Returnează true dacă colecția este numai pentru citire și false altfel

Remove(T) - Înlătură o instanță de tipul T din colecție

Liste generice. Liste sunt foarte asemanatoare cu tablourile, cu avantajul că nu trebuie să decideți în prealabil câte elemente vor contine. Listele implementeaza interfata IList<T> care extinde interfata ICollection <T>. Interfata IList<T> conține membri suplimentari care sunt specifici lucrului cu liste:

- **Item -** Un indexator care obține sau stabilește valoarea pentru un indice dat.
- **IndexOf(T) -** returnează indexul unei instanțe de tipul T, dacă se află în colecție sau -1 altfel.
- **Insert(int, T) -** Inserează o instant de tipul T în colecție la un indice dat.
- **RemoveAt (int) -** Sterge elementul cu indexul specificat din colecție.

Clasa List<T>. Este clasa colectie cea mai frecvent utilizată. Elementele sale sunt conținute într-un tablou, care este redimensionat automat la adaugarea de elemente noi. Clasa are trei constructori:

- **List <T> ()** - Creează o nouă instanță cu capacitate implicită. Adică, tabloul folosit pentru colectarea datelor începe cu o dimensiune implicita și se redimensionează la adăugarea de elemente.
- **List <T> (IEnumerable <T>) -** Creează o nouă instanță care este completată cu elementele din **IEnumerable <T>.**

List <T> (int) - Creează o nouă instanță cu o capacitate inițială specificată.

Clasa List<T> adaugă o multime de metode pentru adăugarea, eliminarea articolelor și extragerea de articole din colecție:

- **AddRange (IEnumerable <T>)** - Adauga conținutul unui IEnumerable <T> la sfarsitul listei
- **GetRange (int, int)** - Returnează o List<T> care conține un subset de elemente din lista
- **InsertRange(int, IEnumerable <string>)** - Inserează conținutul unui IEnumerable <T> în colecție la un index specificat
- **RemoveRange(int, int)** - Elimină un subset de articole din colecție
- **Exist(Predicate (<T>))** - Returnează true dacă există cel puțin un element în listă pentru care predicatul este true
- **Find(Predicațe <T>)** - Returnează primul element din listă pentru care predicatul este true
- **FindAll(Predicate <T>)** - Returnează o lista List<T> care conține toate elementele pentru care predicatul este true
- **FindIndex (Predicate <T>))**
- **FindIndex (int, Predicate <T>)**
- **FindIndex (int, int, Predicate <T>)** - Returnează indexul primului element din lista pentru care predicatul este true
- **FindLast (Predicate <T>)** - Returnează ultimul element din listă pentru care predicatul este true
- **Reverse(); Reverse(int, int) -** Inversează ordinea întregii liste sau a unei regiuni specificate a listei

- **Sort(); Sort(Comparison<T>); Sort(IComparer<T>); Sort(int, int, IComparer<T>)** - Sortează întreaga listă sau o regiune a listei, fie folosind comparatorul de articole implicit, fie utilizând un comparator personalizat

Dicționare generice. Un dicționar este o colectie care stochează perechi cheie/valoare si implementeaza interfata IDictionary<TKey, TVal>. Interfața IDictionary <TKey, TVal> definește comportamentul pentru un dicționar care folosește chei de tipul TKey pentru a stoca valorile de tip Tval. Membrii acestei interfețe sunt:

- **Add (TKey, TVal) -** Adaugă o nouă pereche cheie/valoare.
- **ContainsKey (TKey) –** Returnează true dacă dicționarul conține o cheie specificată.
- **Remove (TKey) –** Sterge perechea cheie/valoarea.
- **Item –** Returneaza un indexator care primește sau setează valoarea asociată unei chei specificate.
- **Keys -** Returnează o ICollection<TKey> care conține toate cheile.
- **Values -** returnează o ICollection <TVal> care conține toate valorile.

Structura KeyValuePair<TKey, TVal> este o structura care conține o cheie și o valoare. Toți membrii interfeței **ICollection<T>** sunt disponibili prin interfața **IDictionary <TKey, TVal>,** unde **T** este **KeyValuePair <TKey, TVal>. KeyValuePair<TKey, Tval>(TKey, TVal)** creaza o instanta cu cheia si valoarea specificate

- **Key –** Returneaza cheia
- **Value -** Returneaza valoarea

Structura KeyValuePair acționează ca un adaptor, care permite unei clase care funcționează cu două tipuri (TKey și TVal) să lucreze cu unul (KeyValuePair <TKey, TVal>).

Clasa Dictionary<TKey, TVal>. Este implementarea interfeței IDictionary <TKey, TVal> și a structurii KeyValuePair <TKey, TVal> si are urmatorii constructori:

- **Dictionary<TKey,TVal>() -** Creează un dicționar cu capacitatea inițială implicită și utilizează comparatorul de egalitate implicit pentru TKey
- **Dictionary<TKey, TVal>(IDictionary<TKey, TVal>)-** Creează un dicționar cu capacitatea inițială implicită și utilizează comparatorul de egalitate implicit pentru TKey, populat cu conținutul IDictionary<TKey, TVal>
- **Dictionary<TKey, TVal>(IEqualityComparer <TKey>) -** Creează un dicționar cu capacitatea inițială implicită și folosind IEqualityComparer<TKey> specificat
- **Dictionary<TKey, TVal>(int) -** Creează un dicționar cu capacitatea inițială specificată

- **Dictionary<TKey, TVal>(IDictionary<TKey, TVal> ,IEqualityComparer<TKey, TVal>) -** Creează un dictionar cu capacitate inițială implicita, utilizeaza comparatorul IEqualityComparer<TKey>, populat cu continutul lui IDictionary<TKey, TVal>
- **Dictionary<TKey, TVal>(int,IEqualityComparer<TKey>)** - Creează un dictionar cu capacitate inițială specificata, utilizeaza comparatorul IEqualityComparer<TKey>

Cele trei opțiuni pe care le putem seta la crearea unui Dictionary <TKey, TVal> sunt capacitatea inițială a colecției, conținutul inițial al colecției și implementarea comparatorului IEqualityComparer <TKey> care va fi utilizată pentru a determina dacă două chei sunt egale.

Metodele clasei Dictionary<TKey, TVal> sunt :

- **ContainsKey(TKey) -** Returneaza true daca colectia contine cheia specificata
- **ContainsValue(TVal)** - Returneaza true daca colectia contine valoarea specificata
- **Comparer** - Returneaza **IEqualityComparer<T>**
- **Count** - Returneaza numarul de perechi key/value din colectie

Multimi Generice. Multimile sunt colecții care nu au elemente duplicate și, în general, elememtele lor nu sunt într-o anumita ordine. Ca instrument de programare, seturile generice .NET sunt utile, deoarece implementeaza operatii cu multimi ca: Compararea colecțiilor de obiecte intre ele, Intersectia, reuniunea, diferenta, incluziunea a două colecții etc.

Interfata ISet<T>. Interfața ISet <T> ne permite să lucram cu diferite implementări de colectii set într-o manieră eficientă. Membrii interfeței ISet <T> sunt:

- **Add (T) -** Adăugă un element la set
- **ExceptWith (IEnumerable <T>) -** Îndepărtează toate elementele continute I IEnumerable <T> din set
- **IntersectWith (IEnumerable <T>) -** Modifică setul astfel încât să conțină doar elementele care sunt de asemenea în IEnumerable <T>
- **IsProperSubsetOf (IEnumerable <T>) –** Returnează true dacă IEnumerable <T> conține unul sau mai multe elemente in plus fata de set
- **IsProperSupersetOf (IEnumerable <T>) –** Returnează true dacă setul are cel puțin un element în plus fata de IEnumerable <T>
- **IsSubsetOf(IEnumerable <T>)** - Intoarce **true** dacă IEnumerable <T> conține toate elementele din set
- **IsSupersetOf(IEnumerable <T>)** - Returnează **true** dacă **set-**ul conține toate elementele din IEnumerable <T>

- **Overlaps(IEnumerable <T>)** – returnează **true** dacă setul și IEnumerable <T> conțin unul sau mai multe elemente în comun
- **SetEquals(IEnumerable <T>)** - Returnează **true** dacă setul și IEnumerable <T> conțin aceleași elemente
- **SymmetricExceptWith(IEnumerable <T>)** - Modifică setul astfel încât să conțină elementele care sunt în set și în IEnumerable <T>, dar nu sunt în ambele
- **UnionWith <IEnumerable <T>)** - Modifică setul astfel încât să conțină atat elementele care sunt în set cat si cele care sunt în IEnumerable <T>

Clasa HashSet <T> este implementarea standard a interfeței ISet <T>. Constructorii pentru HashSet <T> sun:

- **HashSet <T> ()** - Creează un set folosind IEqualityComparer<T> implicit pentru tipul T
- **HashSet <T> (IEnumerable <T>) -** Creează un set folosind IEqualityComparer <T> implicit, populat cu conținutul IEnumerable <T>
- **HashSet <T> (IEqualityComparer <T>)** - Creează un set folosind IEqualityComparer<T> specificat pentru compararea elementelor
- **HashSet<T>(IEnumerable<T>, IEqualityComparer<T>)** - Creează un set folosind IEqualityComparer<T> specificat pentru comparații de elemente și populat cu conținutul IEnumerable <T>

Cozi și stive generice. Sunt utilizate pentru a crea cozi și stive. Ambele tipuri de colecție restricționează accesul la articolele colectate. Cozile vă permit să adăugați elemente la sfârșitul cozii și să luați articole din capul cozii. Disciplina: first-in first-out. Articolele sunt returnate în ordinea în care sunt adăugate. O stivă, în schimb, vă permite să regăsiți numai elementul adăugat cel mai recent: Disciplina: last-in first-out.

Clasa **Queue<T> este o colecție cu disciplina FIFO (first-in, first-out) si are urmatorii constructori:**

- **Queue <T> ()** - Creează o coadă goală cu capacitatea inițială implicită
- **Queue <T> (IEnumerable <T>) -** Creează o coadă populată cu conținutul din IEnumerable <T>
- **Queue <T> (int) -** Creează o coadă goală cu capacitatea specificată

Observam că nu există constructori care să admita implementări IEqualityComparer<T> sau IComparer<T>. Acest lucru se datorează faptului că Queue <T> va colecta articole duplicate.

Membri clasei **Clasa Queue<T>** sunt :

- **Enqueue (T)** - Adăugă un element la sfârșitul cozii
- **Dequeue ()** - Înlătură și returnează elemental de la începutul cozii
- **Peek ()** - returnează elemental de la începutul cozii fără a-l elimina
- **Count** - Returnează numărul de articole din coadă

Când utilizați **Queue <T>**, puteți lucra doar cu capul și coada cozii. Trebuie să utilizați metodele **Enqueue, Dequeue** și **Peek** pentru a modifica conținutul unei **Queue <T>**

Clasa Stack<T>. Stivele sunt colecții last-in, first-out (LIFO). Adăugarea unui articol la o stivă se numește pushing, iar regăsirea unui articol se numește popping. Elementul cel mai recent adaugat într-o stivă este cel care este returnat la prima scoatere din stiva. Constructorii pentru Stack <T> sunt :

- **Stack<T>() -** Creaza o stiva vida cu capacitatea initiala impliceta.
- **Stack<T>(IEnumerable<T>) -** Creaza o stiva vida cu capacitatea initiala impliceta cu continutul din IEnumerable<T>
- **Stack<T>(int) -** Creaza o stiva vida cu capacitatea specificata.

Când lucram cu **Stack <T>**, avem acces doar la sfârșitul colecției. Putem adăuga un element nou folosind metoda **Push**, putam obtine ultimul element adaugat folosind metoda **Pop** și putem accesa articolul cel mai recent, fără a-l elimina, folosind metoda **Peek**.

- **Push (T)** - Adaugă un nou element la stivă
- **Pop ()** - Îndepărtează și returnează cel mai recent articol adăugat din stivă
- **Peek ()** - returnează cel mai recent articol adăugat fără a-l elimina

Tablourille sunt Colectii. System.Array este clasa de bază utilizată pentru a susține tablouri. Nu putem deriva din clasa System.Array, dar putem profita de caracteristica clasei System.Array, iar una dintre cele mai utile dintre aceste caracteristici este că System.Array implementează interfetele IList <T>, ICollection <T> și IEnumerable <T> . Aceasta înseamnă că tablourile sunt și colecții și pot fi utilizate alături de celelalte clase de colecții generice. În toate cazurile, tablourile trebuie să fie create folosind facilitatile normale ale limbajului C# și să trecem tabloul ca argument la metoda System.Array statică pe care dorim să o utilizam.

Membrii clasei **Clasa System.Array** sunt:

- **BinarySearch (T[],T); BinarySearch (T [],T,IComparer<T>)** Caută in tablou o valoare, fie folosind **IComparer<T>** implicit fie utilizând un **IComparer<T>** personalizat.
- **ConvertAll (TIn [], TOut [], Converter <TIn, TOut>)** Convertește elementele dintr-un tablou într-un tip diferit

- **Copy (T [], T []) -** Copiază elementele dintr-un tablou într-un tablou diferit de același tip
- **Exists (T [], Predicat <T>) -** Returnează adevărat dacă cel puțin un element din tablă îndeplinește condițiile de predicat
- **Find (T [], Predicați <T>); FindLast (T [], Predicate <T>)** Găsește primul sau ultimul element din tablou care îndeplinește condițiile de predicat
- **FindAll (T [], Pridcate <T>) -** Returnează toate elementele din tablou care îndeplinesc condițiile de predicat ca un nou tablou de tip T
- **FindIndex(T[],Predicate<T>); FindLastIndex(T[], Predicate<T>) -** Returnează indexul primului sau ultimului articol din tablou care se potriveste cu condițiile de predicat
- **Sort (T []); Sort (T [], IComparer <T>) -** Sortează tabloul în acelas spatiu, folosind fie comparatorul implicit pentru T, fie IComparer-ul specificat
- **TrueForAll (T [], Predicate <T>) -** Returnează adevărat dacă toate elementele din tablou îndeplinesc condițiile de predicat

Utilizarea proprietăților implementate automat. Facilitatile de proprietate C# vă permit să definiți date dintr-o clasă într-un mod care decuplează datele de modul în care sunt setate și preluate. Exemplul 1 conține o astfel de clasă numită Produs.

Exemplul 1. Defining a property

```
public class Product {
        private string name;

        public string Name {
                get { return name; }
                set { name = value; }
        }
}
```

Instrucțiunile din blocul de cod get (cunoscut sub numele de getter) sunt efectuate atunci când valoarea proprietății este citită, iar instrucțiunile din blocul de cod set (cunoscut sub numele de setter) sunt efectuate atunci când o valoare este atribuită proprietății. O proprietate este utilizată de alte clase ca și cum ar fi un câmp așa cum este arătat în exemplul 2, care ilustrează modul în care folosim proprietatea.

Exemplul 2. Consuming a property

```
        protected string GetMessage() {
                Product myProduct = new Product();
                myProduct.Name = "Kayak";
                return String.Format("Product name: {0}", myProduct.Name);
        }
```

Putem vedea că valoarea proprietății este utilizata la fel ca un câmp obișnuit. Utilizarea proprietăților este preferabilă folosirii câmpurilor, deoarece puteți modifica instrucțiunile din blocurile get și set, fără a fi nevoie să modificați toate clasele care depind de proprietate.

Proprietatile specificate astfel sunt stufoase inutil, așa cum puteți vedea în exemplul 3.

Exemplul 3. Verbose property definitions

```
public class Product {
        private int productID;
        private string name;
        private string description;
        private decimal price;
        private string category;
        public int ProductID {
                get { return productID; }
                set { productID = value; }
        }
        public string Name {
                get { return name; }
                set { name = value; }
        }
        public string Description {
                get { return description; }
                set { description = value; }
        }
        public decimal Price {
                get { return price; }
                set { price = value; }
        }
        public string Category {
                get { return category; }
                set { category = value; }
        }
}
```

Soluția pentru simplificarea codului este utilizarea proprietăților implementate automat, cunoscute și sub numele de proprietăți automate. Cu o proprietate automată, puteți crea modelul unei proprietăți cu un câmp fără cod redundant, așa cum arată exemplul 4.

Exemplul 4. Using automatically implemented properties

```
public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
```

```
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
```

Utilizarea unei proprietăți automate nu diferă de utilizarea unei proprietăți obișnuite; codul din clasa de cod din exemplul 2 va funcționa fără nicio modificare. Folosind proprietăți automate, creăm cod care este mai ușor de citit și păstrăm în continuare flexibilitatea oferită de proprietăți. Dacă vreodată trebuie să schimbăm modul în care este implementată o proprietate, putem reveni la formatul proprietății obișnuite. Să ne imaginăm că trebuie să schimbăm modul în care este administrată proprietatea Name, așa cum se arată în exemplul 5.

Exemplul 5. Reverting from an automatic to a regular property

```
    public class Product {
        private string name;
        public int ProductID { get; set; }
        public string Name {
            get {
                return ProductID + name;
            }
            set {
                name = value;
            }
        }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
```

Utilizarea inițializatorilor de obiecte și colecții. O altă sarcină obișnuită de programare este construirea unui obiect nou, iar apoi trebuie să atribuim separate valori proprietăților, așa cum este ilustrat în exemplul 6, care arată modificările pe care le-am făcut la clasa de cod Default.aspx.cs.

Exemplul 8. Constructing and initializing an object with properties

```
        protected string GetMessage() {
            // create a new Product object
            Product myProduct = new Product();
            // set the property values
            myProduct.ProductID = 100;
            myProduct.Name = "Kayak";
            myProduct.Description = "A boat for one person";
            myProduct.Price = 275M;
            myProduct.Category = "Watersports";
            return String.Format("Category: {0}", myProduct.Category);
```

```
}
```

Trebuie să parcurgem trei etape pentru a crea un obiect de produs și pentru a produce un rezultat: creăm obiectul, setăm valorile parametrilor și apoi returnăm valoarea.

Putem elimina unul dintre acești pași folosind funcția de inițializare a obiectului C#, care ne permite să creăm și să populăm instanța Produs într-o singură etapă, așa cum se arată în exemplul 7.

Exemplul 7. Using the object initializer feature

```
protected string GetMessage() {
        // create a new Product object
        Product myProduct = new Product {
                ProductID = 100, Name = "Kayak",
                Description = "A boat for one person",
                Price = 275M, Category = "Watersports"
        };
        return String.Format("Category: {0}", myProduct.Category);
}
```

Acoladele care urmeaza instantierea contin inițializatorul, pe care îl utilizăm pentru a furniza valori parametrilor ca parte a procesului de construcție.

Aceeași caracteristică ne permite să inițializăm conținutul colecțiilor și tablourilor ca parte a procesului de construcție, așa cum se arată în exemplul 8.

Exemplul 8. Initializing collections and arrays

```
protected string GetMessage() {
        string[] stringArray = { "apple", "orange", "plum" };
        List<int> intList = new List<int> { 10, 20, 30, 40 };
        Dictionary<string, int> myDict = new Dictionary<string, int> {
        { "apple", 10 }, { "orange", 20 }, { "plum", 30 }
        };

        return String.Format("Fruit: {0}", (object)stringArray[1]);
}
```

Listarea rezultatului demonstrează modul de construire și inițializare a unui tablou și a două clase din biblioteca de colecții generice.

Utilizarea metodelor de extensie. Metodele de extensie sunt o modalitate convenabilă de a adăuga metode la clase pe care nu le deții și nu le puteți modifica direct. Exemplul 9 arată clasa ShoppingCart, pe care am definit-o într-un nou fișier de clasă numit ShoppingCart.cs. ShoppingCart reprezintă o colecție de obiecte de tip **Products**.

Exemplul 9. The ShoppingCart class

```
using System.Collections.Generic;
namespace LanguageFeatures {
```

```
        public class ShoppingCart {
                public List<Product> Products { get; set; }
        }
}
```

Să presupunem că trebuie să putem determina valoarea totală a obiectelor Product din clasa ShoppingCart, dar nu putem modifica clasa în sine, poate pentru că provine de la o terță parte și nu avem codul sursă. Putem folosi o metodă de extensie pentru a obține funcționalitatea de care avem nevoie. Exemplul 10 arată clasa MyExtensionMethods, pe care am definit-o într-un nou fișier de clasă numit MyExtensionMethods.cs.

Exemplul 12. Defining an extension method

```
namespace LanguageFeatures {
        public static class MyExtensionMethods {
                public static decimal TotalPrices(this ShoppingCart cartParam) {
                        decimal total = 0;
                        foreach (Product prod in cartParam.Products) {
                                total += prod.Price;
                        }
                        return total;
                }
        }
}
```

Metodele de extensie trebuie să fie statice și definite într-o clasă statică. Cuvânt cheie **this** din fața primului parametru marchează TotalPrices ca o metodă de extensie. Primul parametru spune sistemului .NET la ce clasă se poate aplica metoda de extensie - ShoppingCart în cazul nostru. Ne putem referi la instanța ShoppingCart la care s-a aplicat metoda de extensie folosind parametrul cartParam. Metoda noastră enumeră produsele prin ShoppingCart și returnează suma proprietății Product.Price. Exemplul 11 arată modul în care aplicăm o metodă de extensie într-o alta clasa.

Metodele de extensie nu permit încălcarea regulilor de acces pe care le definesc clasele pentru metodele, câmpurile și proprietățile lor. Putem extinde funcționalitatea unei clase folosind o metodă de extensie, dar folosind doar membrii clasei la care am avut acces oricum.

Exemplul 11. Applying an extension method

```
                protected string GetMessage() {
                        ShoppingCart cart = new ShoppingCart {
                                Products = new List<Product> {
                                        new Product {Name = "Kayak", Price = 275M},
                                        new Product {Name = "Lifejacket", Price = 48.95M},
                                        new Product {Name = "Soccer ball", Price = 19.50M},
                                        new Product {Name = "Corner flag", Price = 34.95M}
                                }
```

```
            };
            decimal cartTotal = cart.TotalPrices();
            return String.Format("Total: {0:c}", cartTotal);
        }
```

După cum puteți vedea, apelam metoda TotalPrices pe un obiect ShoppingCart ca și cum ar face parte din clasa ShoppingCart, chiar dacă este o metodă de extensie definită de o altă clasă .NET va găsi clasele dvs. de extensie dacă se află în același spațiu de nume ca si clasa în care apelam metoda de extensie sau într-un spațiu de nume care face obiectul unei instrucțiuni **using**.

Aplicarea metodelor de extensie la o interfață. De asemenea, putem crea metode de extensie care se aplică unei interfețe, ceea ce ne permite să apelăm metoda de extensie din toate clasele care implementează interfața. Exemplul 12 arată clasa ShoppingCart actualizată pentru a implementa interfața IEnumerable <Product>.

Exemplul 14. Implementing an interface in the ShoppingCart class

```
public class ShoppingCart : IEnumerable<Product> {
    public List<Product> Products { get; set; }
    public IEnumerator<Product> GetEnumerator() {
        return Products.GetEnumerator();
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

Acum putem actualiza metoda noastră de extensie, astfel încât să functioneze la nivelul claselor care implementeaza IEnumerable <Product>, așa cum se arată în exemplul 13.

Exemplul 13. An extension method that works on an interface

```
public static class MyExtensionMethods {
    public static decimal TotalPrices(this IEnumerable<Product> productEnum) {
        decimal total = 0;

        foreach (Product prod in productEnum) {
            total += prod.Price;
        }
        return total;
    }
}
```

Am schimbat tipul parametrului în IEnumerable <Product>, ceea ce înseamnă că bucla de foreach din corpul metodei funcționează direct pe obiecte de produs. Trecerea la interfață înseamnă că putem calcula valoarea totală a obiectelor Produs enumerate de orice

IEnumerable <Product>, care include cazuri de ShoppingCart, dar și tablouri de obiecte de Produs, așa cum se arată în exemplul 14.

Exemplul 14. Applying an extension method to different implementations of the same interface

```
protected string GetMessage() {
        IEnumerable<Product> products = new ShoppingCart {
                Products = new List<Product> {
                        new Product {Name = "Kayak", Price = 275M},
                        new Product {Name = "Lifejacket", Price = 48.95M},
                        new Product {Name = "Soccer ball", Price = 19.50M},
                        new Product {Name = "Corner flag", Price = 34.95M}
                }
        };
        Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M},
                new Product {Name = "Corner flag", Price = 34.95M}
        };
        decimal cartTotal = products.TotalPrices();
        decimal arrayTotal = products.TotalPrices();
        return String.Format("Cart Total: {0:c}, Array Total: {1:c}",
                cartTotal, arrayTotal);
}
```

Utilizarea expresiilor Lambda. Putem utiliza un delegat pentru a face o metoda de filtrare FilterByCategory mai generală. În acest fel, delegatul care va fi invocat pentru fiecare Produs poate filtra obiectele în orice mod pe care l-am alege, așa cum este ilustrat de exemplul 15, care arată cum putem adăuga o metodă de extensie numită Filtru la clasa MyExtensionMethods.

Exemplul 15. Using a delegate in an extension method

```
public static class MyExtensionMethods {
        public static decimal TotalPrices(this IEnumerable<Product> productEnum) {
                decimal total = 0;
                foreach (Product prod in productEnum) {
                        total += prod.Price;
                }
                return total;
        }
        public static IEnumerable<Product> FilterByCategory(
                this IEnumerable<Product> productEnum, string categoryParam) {
                foreach (Product prod in productEnum) {
                        if (prod.Category == categoryParam) {
                                yield return prod;
```

```
                }
            }
        }
        public static IEnumerable<Product> Filter(
        this IEnumerable<Product> productEnum, Func<Product, bool> selectorParam)
        {
            foreach (Product prod in productEnum) {
                if (selectorParam(prod)) {
                    yield return prod;
                }
            }
        }
}
```

Am folosit un delegate standard Func ca parametru de filtrare, ceea ce înseamnă că nu trebuie să mai definim tipul delegatului. Delegatul primeste un parametru de tip **Product** și returnează un boolean, care va fi true dacă respectivul produs trebuie inclus în rezultate. Exemplul 16, arată modul de utilizare a metodelor de extensie pentru filtrare.

Exemplul 16. Using the filtering extension method with a func

```
protected string GetMessage() {
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category =
            "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category =
            "Watersports",
            Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer",
            Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer",
            Price = 34.95M}
        }
    };
    Func<Product, bool> categoryFilter = delegate(Product prod) {
    return prod.Category == "Soccer";
    };
    decimal total = products.Filter(categoryFilter).TotalPrices();
    return String.Format("Soccer Total: {0:c}", total);
}
```

Acum putem filtra obiectele **Product** folosind orice criterii pe care le specificam în delegat, dar trebuie să definim un Func pentru fiecare tip de filtrare dorit, ceea ce nu este ideal. Alternativa

mai puțin labrioasă este utilizarea unei expresii lambda, care este un format concis pentru exprimarea unui corp de metodă într-un delegat. Putem folosi o lambda expresie pentru a înlocui definiția delegatului nostru, așa cum se arată în exemplul 17.

Exemplul 17. Using a lambda expression to replace a delegate definition

```
...
protected string GetMessage() {
IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports",
                Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer",
                Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer",
                Price = 34.95M}
        }
};

Func<Product, bool> categoryFilter = prod => prod.Category == "Soccer";
decimal total = products.Filter(categoryFilter).TotalPrices();
return String.Format("Soccer Total: {0:c}", total);
}
...
```

Parametrul este exprimat fără a specifica un tip, care va fi dedus automat. Putem scrie sintaxa noastră și mai simplu, eliminând declararea funcției în întregime, așa cum se arată în exemplul 18.

Exemplul 18. A lambda expression without a func

```
...
protected string GetMessage() {
        IEnumerable<Product> products = new ShoppingCart {
                Products = new List<Product> {
                        new Product {Name = "Kayak",Category ="Watersports",Price = 275M},
                        new Product {Name = "Lifejacket", Category = "Watersports",
                                Price = 48.95M},
                        new Product {Name = "Soccer ball", Category = "Soccer",
                                Price = 19.50M},
                        new Product {Name = "Corner flag", Category = "Soccer",
                                Price = 34.95M}
                }
        };
        decimal total = products.Filter(prod => prod.Category == "Soccer").TotalPrices();
        return String.Format("Soccer Total: {0:c}", total);
```

```
}
...
```

În acest exemplu, am furnizat expresia lambda ca parametru pentru metoda Filter. Acesta este un mod frumos și natural de exprimare a filtrului pe care dorim să-l aplicăm.

Inferența automata a tipurilor. Cuvântul cheie **var** in limbajul C# vă permite să definiți o variabilă locală fără a specifica explicit tipul variabilei, așa cum este demonstrat în exemplul 19.

Aceast mecanism se numește inferență de tip sau tipizare implicita.

Exemplul 19. Using type inference

```
...
var myVariable = new Product { Name = "Kayak", Category = "Watersports", Price = 275M };
string name = myVariable.Name; // legal
int count = myVariable.Count; // compiler error
...
```

Aici MyVariable nu are un tip definit si prin urmare compilatorul va deduce tipul din cod. Dupa cum vedem compilatorul va permite apelarea numai a membrilor clasei inferioare - **Product** în acest caz .

Utilizarea tipurilor anonime. Combinând inițializatoarele de obiecte și inferența de tip, putem crea obiecte simple de stocare a datelor fără a fi nevoie să definim clasa sau structura corespunzătoare. Exemplul 20 arată ecest lucru.

Exemplul 20. Creating an anonymous type

```
protected string GetMessage() {
        var myAnonType = new {
                Name = "Kayak",
                Category = "Watersports"
        };
        return string.Format("Name: {0}, Type: {1}", myAnonType.Name,
                myAnonType.Category);
}
```

În acest exemplu, myAnonType este un obiect tipizat anonim. Aceasta nu înseamnă că tipizarea este dinamică ci doar că definiția tipului va fi dedusă automat de compilator. Puteți obține și seta doar proprietățile care au fost definite în inițializator, de exemplu.

Compilatorul C# generează clasa pe baza numelui și tipului parametrilor din inițializator. Două obiecte tipizate anonim care au aceleași nume și tipuri de proprietăți vor fi alocate aceleiași clase generate automat. Aceasta înseamnă că putem crea tablouri de obiecte tipizate anonim, așa cum putem vedea în exemplul 21.

Exemplul 21. Creating an array of anonymously typed objects

```
protected string GetMessage() {
        var oddsAndEnds = new[] {
                new { Name = "Blue", Category = "Color"},
                new { Name = "Hat", Category = "Clothing"},
                new { Name = "Apple", Category = "Fruit"}
        };
        StringBuilder result = new StringBuilder();
        foreach (var item in oddsAndEnds) {
                result.Append(item.Name).Append(" ");
        }
        return result.ToString();
}
```

Observați că folosim var pentru a declara tabloul. Trebuie să facem acest lucru pentru că nu avem un tip specificat așa cum avem in cazul unui tablou de un tip predefinit. Chiar dacă nu am definit o clasă pentru niciunul dintre aceste obiecte, putem totuși enumera conținutul tabloului și atribui valori proprietatilor. Acest lucru este important deoarece, fără această caracteristică, nu am putea să creăm tablouri cu obiecte tipizate anonim. Sau, mai degrabă, am putea crea tablourile, dar nu am fi capabili să facem nimic util cu ele.

Utilizarea tipurilor generice. Tipurile generice ne permit să definim clase care operează pe tipuri variabile. Frecvent, avem nevoie sa putem crea clase care să funcționeze pe o serie de tipuri diferite. Aceasta necessitate este rezolvata de tipurile generice. În exemplul 22, avem definita o astfel de clasă în fișierul MyContainers.cs.

Exemplul 22. Defining the ValueContainer class

```
namespace LanguageFeatures {
        public class ValueContainer<T> {
                public T Value { get; set; }
                public bool HasValue {
                        get { return Value != null; }
                }
        }
}
```

Această clasă se numește ValueContainer <T>. Specificatia <T> indică faptul că această clasă are un parametru de tip generic numit T. Un parametru de tip generic spune compilatorului că dorim să lucrăm cu un tip specific, dar nu știm care va fi încă, așa că ne referim la acesta ca T.

Convenția este ca pentru tipurile generice sa folosim litera unică T sau un nume descriptiv prefixat cu T, cum ar fi TKey sau TValue.

Mapam tipul generic la un tip specific atunci când instantiem clasa ValueContainer. Putem vedea acest lucru în exemplul 23.

Exemplul 23. Instantiating a class with a generic type parameter

```
protected string GetMessage() {
        ValueContainer<string> stringContainer=new valueContainer<string>();
        stringContainer.Value = "Hello";
        return String.Format("Year: {1}", stringContainer.Value);
}
```

Când creăm o instanță a clasei ValueContainer<T>, înlocuim T cu tipul pe care dorim să-l utilizăm în paranteze unghiulare. Vrem să creăm o instanță ValueContainer<T> care să funcționeze pe obiecte string, așa că am creat un obiect ValueContainer <string>.

In exemplul 24 vedem modul de utilizare a clasei noastre tipizaată generic.

Exemplul 24. Rewriting the initial code using a generically typed class

```
protected string GetMessage() {
        ValueContainer<string> stringContainer=new valueContainer<string>();
        stringContainer.Value = "Hello";
        ValueContainer<DateTime> dtContainer=new ValueContainer<DateTime>();
        dtContainer.Value = DateTime.Now;

        if (stringContainer.HasValue && dtContainer.HasValue) {
                return String.Format("Char: {0}, Year: {1}",
                stringContainer.Value.ToCharArray().First(),
                dtContainer.Value.Year);
        } else {
                return "No values";
        }
}
```

LINQ (Language Integrated Query) este o componenta a platformei Microsoft .NET care adauga suport pentru interogari de date direct din sintaxa limbajelor .NET. Această componentă defineşte un set de operatori de interogare care pot fi folositi pentru chestionarea, proiectarea si filtrarea datelor din colectii, clase numarabile, fisiere XML, baze de date relaţionale şi alte surse de date, cu condiţia ca informaţia sa fie încapsulată în obiecte. Rezultatul unei interogari este o colectie de obiecte stocate în memoria programului care pot fi enumerate folosind o funcţie de parcurgere standard cum ar fi funcţia foreach din C#. LINQ este o facilitate importanta si convingătoare adaugata la .NET.

LINQ are o sintaxă asemănătoare cu SQL pentru interogarea datelor din clase, care se bazeaza pe cuvinte cheie ca:

from <specifică colecțiile relevante pentru a răspunde la interogare și elementele de identificare pentru a itera peste aceste colecții>

where <specifica conditiile de filtrare>

select <structura de date care trebuie returnata>

LINQ folosește, de asemenea, expresiile familiare cu notația punct pentru a avea acces la proprietățile sau metodele de extensie asociate cu obiectele.

source.**Where**(<conditii>).**Select**(<structura de date>);

Sa presupunem că avem o colecție de obiecte de tip Product și dorim să găsim și să afișăm cele mai mari trei prețuri. Utilizand sintaxa LINQ putem scrie acest lucru ca in exemplul 25.

Exemplul 25. Using LINQ to query data

```
...
protected string GetMessage() {
        Product[] products = {
                new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
                new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
                new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
                new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        var foundProducts = from match in products
                                orderby match.Price descending
                                select match.Price;
        int count = 0;
        StringBuilder result = new StringBuilder();
                foreach (var price in foundProducts) {
                result.AppendFormat("Price: {0} ", price);
                if (++count == 3) {
                break;
                }
        }
        return result.ToString();
}
...
```

Puteți vedea ca interogarea este asemănătoare cu SQL. Ordonăm obiectele Produs în ordine descrescătoare și folosim cuvântul cheie select pentru a returna doar valorile proprietății Preț. Acest stil de LINQ este cunoscut sub numele de sintaxa de interogare (query syntax) și este cel pe care dezvoltatorii îl găsesc cel mai confortabil atunci când încep să folosească LINQ.

Alternativa la aceasta sintaxa este sintaxa dot-notation syntax, sau dot notation, care se bazează pe metode de extensie. Exemplul 26 arată cum putem folosi această sintaxă alternativă pentru a procesa obiectele de tip Product.

Exemplul 26. Using LINQ dot notation

```
...
protected string GetMessage() {
        Product[] products = {
                new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
                new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
                new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
                new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        var foundProducts = products.OrderByDescending(e => e.Price)
                                    .Take(3)
                                    .Select(e => e.Price);
        StringBuilder result = new StringBuilder();
        foreach (var price in foundProducts) {
                result.AppendFormat("Price: {0} ", price);
        }
        return result.ToString();
}
...
```

Desi această interogare LINQ, nu este la fel de plăcută ochiului ca cea exprimată prin sintaxa de interogare, nu toate funcțiile LINQ au cuvinte cheie in sintaxa de interogare si de aceia pentru programarea LINQ serioasă, trebuie să trecem la metode de extensie. Fiecare dintre metodele de extensie LINQ din Exemplul 26 este aplicată unui IEnumerable <T> și returnează tot un IEnumerable <T>, ceea ce ne permite să îmbricam metodele pentru a forma interogări complexe.

O mulțime de metode de extensie LINQ se află în spațiul de nume System.Linq, pe care trebuie introdus în aplicatie cu o declarație **using** înainte de a putea face interogări. Visual Studio adaugă automat spațiul de nume la clasele din spatele codurilor și le face disponibile pentru utilizare în aplicatie.

PLINQ este o implementare paralelă a limbajului LINQ to objects care permite programatorului să efectueze interogări LINQ în paralel, astfel încât mai multe elemente de date să fie procesate simultan. PLINQ funcționează cu aceleași surse de date pe care le utilizează LINQ pentru obiecte, și anume IEnumerable < > și IEnumerable.

Exemplul 27. Parallel Language Integrated Query

```
namespace Exemplul_7 {

   class Listing_01 {
```

```
    static void myMain() {
        int[] sourceData = new int[100];
        for (int i = 0; i < sourceData.Length; i++) {
            sourceData[i] = i;
        }
        IEnumerable<int> results =
            from item in sourceData.AsParallel()
            where item % 2 == 0
            select item;
    }
}
}
```

C# conține câteva cuvinte cheie specifice LINQ-ului, cum ar fi:

from, **where** si **select**.

Aceste cuvinte cheie sunt mapate la metodele de extensie din clasa Enumerable, astfel că o interogare cum ar fi următoarea:

```
IEnumerable<int> result =
        from item in sourceData
        where (item % 2 == 0)
        select item;
```

este mapat la metodele de extensie corespunzatoare:

```
IEnumerable<int> result =
sourceData
        .Where(item => item % 2 == 0)
    .Select(item => item)
```

Folosirea metodelor de extensie în locul cuvintelor cheie poate fi utilă, deoarece putem utiliza expresii lambda complexe.

Metodele de extensie sunt adesea denumite operatori de interogare standard și se folosesc aplicând notația cu punct.

PLINQ functioneaza pe două clase publice: **ParallelEnumerable-ParallelQuery.**

ParallelEnumerable - conține metode de extensie care funcționează pe tipul **ParallelQuery**.

ParallelEnumerable contine metode paralele corespunzatoare celor din Enumerable, plus altele care permit constructii specifice ParallelQuery.

AsParallel(), este cheia utilizării PLINQ. Conversia unui IEnumerable intr-un ParallelQuery. Pentru a utiliza PLINQ, trebuie să creăm o instanță ParallelQuery apelând metoda AsParallel () pe o instanță a IEnumerable și folosindu-o ca bază pentru aplicarea caracteristicilor LINQ.

AsSequential() este opusa lui AsParallel() și ne permite să aplicăm selectiv paralelism într-o interogare complexă. Conversia unui ParallelQuery intr-un IEnumerable.

Urmatoarele metode configurează instanțele ParallelQuery produse de AsParallel() pentru a schimba modul în care o interogare este executată sau se comportă.

AsOrdered() modifica un ParallelQuery pentru a păstra ordinea de prelucrare a articolelor.
AsUnordered() modifica un ParallelQuery pentru a renunța la ordinea de prelucrare a articolelor.

WithCancellation() modifica ParallelQuery pentru a monitoriza un token de anulare.

WithDegreeOfParallelism() modificați un ParallelQuery pentru a seta o limită superioară a numărului de taskuri utilizate pentru executarea unei interogări.

Creaza o instanță ParallelQuery pe o instanță a IEnumerable, ca bază pentru aplicarea caracteristicilor LINQ

Exemplul 28. AsParallel()

```
int[] sourceData = new int[10];
 for (int i = 0; i < sourceData.Length; i++) {
    sourceData[i] = i;
 }
// define a sequential linq query
 IEnumerable<double> results1 =
    from item in sourceData
    select Math.Pow(item, 2);
```

```
// define a parallel linq query
IEnumerable<double> results2 =
   from item in sourceData.AsParallel()
   select Math.Pow(item, 2);
```

Exemplul 29. Filtrarea Datelor

```
// create some source data
int[] sourceData = new int[100000];
for (int i = 0; i < sourceData.Length; i++) {
   sourceData[i] = i;
}
// define a filtering query using keywords
IEnumerable<double> results1
   = from item in sourceData.AsParallel()
     where item % 2 == 0
     select Math.Pow(item, 2);
// define a filtering query using extension methods
IEnumerable<double> results2
   = sourceData.AsParallel()
   .Where(item => item % 2 == 0)
   .Select(item => Math.Pow(item, 2));
```

Rezultatele obținute de interogarea paralelă nu sunt în aceeași ordine cu cele ale interogării secvențiale.

PLINQ partitioneaza datele sursă pentru a îmbunătăți eficiența taskurilor și acest lucru distruge ordinea naturală a elementelor.

Putem păstra ordinea într-o interogare PLINQ utilizând metoda de extensie AsOrdered(), care modifică instanța ParallelQuery.

Acest lucru reduce performanta. De aceia ar trebui să utilizam metoda AsOrdered() numai dacă ordinea rezultatelor în raport cu datele sursă este importantă.

Exemplul 30. AsOrdered()

```
// create some source data
        int[] sourceData = new int[10];
        for (int i = 0; i < sourceData.Length; i++) {
           sourceData[i] = i;
        }
        // preserve order with the AsOrdered() method
        IEnumerable<double> results =
           from item in sourceData.AsParallel().AsOrdered()
           select Math.Pow(item, 2);
```

Putem controla ordonarea, combinând metodele de extensie

AsOrdered() și **AsUnordered()**.

Metoda AsUnordered() este exact opusa lui AsOrdered() și deci, spune lui PLINQ că ordinea nu trebuie păstrată.

Metoda **Take(n)** ia primele n elemente din sursa de date.

În exemplu, primele zece elemente sunt preluate din sursa de date (Take(10)) după ce este apelată metoda AsParallel(). Pentru această parte a interogării, elementele sunt ordonate.

Elementele luate sunt apoi utilizate ca bază pentru un apel la Select(), care poate fi efectuat fără a pastra ordinea, așa că apelam metoda AsUnordered().

Exemplul 31. AsUnordered()

```
// create some source data
        int[] sourceData = new int[10000];
        for (int i = 0; i < sourceData.Length; i++) {
           sourceData[i] = i;
        }
        // define a query that has an ordered subquery
```

```
var result =
   sourceData.AsParallel().AsOrdered()
   .Take(10).AsUnordered()
   .Select(item => new {
      sourceValue = item,
      resultValue = Math.Pow(item, 2)
   });
```

Puteți cere ca o interogare să fie efectuată paralel utilizând metoda de extensie WithExecutionMode (), care ia o valoare din enumerarea ParallelExecutionMode ca argument.

Valorile enumerării sunt:

Default - PLINQ va decide dacă executia interogari va fi secvențiala sau paralela.

ForceParallelism - Interogarea va fi paralelizată, chiar dacă executia paralelă va fi mai costisitoare decat exacutia secventiala.

Metoda WithExecutionMode() modifică o instanță ParallelQuery;

Exemplul32. WithExecutionMode()

```
// create some source data
int[] sourceData = new int[10];
for (int i = 0; i < sourceData.Length; i++) {
   sourceData[i] = i;
}
// define the query and force parallelism
IEnumerable<double> results =
   sourceData.AsParallel()
 .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
   .Where(item => item % 2 == 0)
   .Select(item => Math.Pow(item, 2));
```

Puteți solicita o limită superioară a numărului de Taskuri care vor fi utilizate pentru a efectua o interogare LINQ utilizând metoda extensiei WithDegreeOfParallelism().

Motorul PLINQ poate alege să utilizeze mai puține sarcini decât sunt specificate prin această metodă dar nu mai multe.

Exemplul urmator demonstrează utilizarea acestei metode pentru a specifica maxim două sarcini.

Exemplul33. WithDegreeOfParallelism()

```
// create some source data
    int[] sourceData = new int[10];
    for (int i = 0; i < sourceData.Length; i++) {
       sourceData[i] = i;
    }
    // define the query and force parallelism
    IEnumerable<double> results =
       sourceData.AsParallel()
       .WithDegreeOfParallelism(2)
       .Where(item => item % 2 == 0)
       .Select(item => Math.Pow(item, 2));
```

Metoda de extensie **AsSequential()** transforma ParallelQuery intr-o sarcina secventiala IEnumerable. Această metodă este opusa metodei AsParallel().

Putem utiliza AsParallel() alternativ cu AsSequential() pentru a activa și dezactiva paralelismul în subinterogari.

Această tehnică poate fi utilă dacă dorim să evitam în mod explicit costurile paralelismului pentru o parte a unei interogări.

Exemplul urmator demonstrează utilizarea acestei metode.

Prima parte a interogării păstrează valorile sursă în paralel, iar a doua parte a interogării, unde valorile pătratului sunt dublate, se efectuează secvențial.

Exemplul 34. AsSequential()

```
// create some source data
        int[] sourceData = new int[10];
        for (int i = 0; i < sourceData.Length; i++) {
           sourceData[i] = i;
        }
        // define the query and force parallelism
        IEnumerable<double> results =
           sourceData.AsParallel()
           .WithDegreeOfParallelism(2)
           .Where(item => item % 2 == 0)
           .Select(item => Math.Pow(item, 2))
           .AsSequential()
           .Select(item => item * 2);
```

Exercitii.

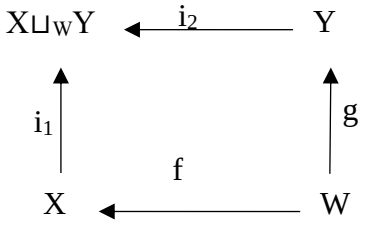
1. Studiati si testati exemplele din textul de mai sus.
2. Pushout :

Daca avem trei multimi X,Y si W si doua functii :

f: W$\to$X si g: W$\to$Y. Atunci o multime P se numeste pushoutul lui X cu Y peste W (sau pushoutul lui f cu g) daca este izomorfa cu multimea:

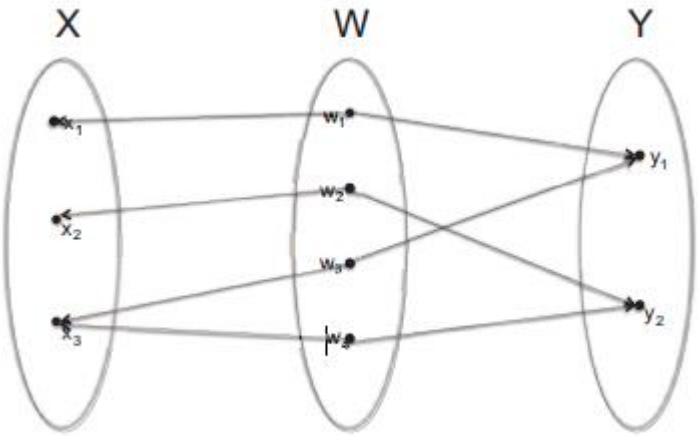
$X \sqcup_W Y = (X \sqcup Y)/\rho$, unde ρ este o relatie de echivalenta generata de functiile f si g dupa cum urmeaza: $x \rho y \Leftrightarrow \exists w \in W$ a, i. $x=f(w)$ si $y= g(w)$.

Diagrama:



se numeste diagrama pushout.

De exemplu daca functiile f:W→X si g:W→Y sunt cele din diagrama



Atunci pushoutul lui f cu g are un singur element pentru ca:
x_1 ρ w_1 ρ y_1 ρ w_3 ρ x_3 ρ w_4 ρ y_2 ρ w_2 ρ x_2.
Pushoutul poate fi folosit pentru a caracteriza epimorfismele: Un morfism f: W→ X este un epimorfism dacă și numai dacă pushoutul lui f și f există și este W.

Vom considera o clasa abstracta PushoutAbstract definite astfel.

```
public interface functie<Td, Tc>
{
    Tc Calcul(Td intrare);
}

public abstract class PushoutAbstract<T1, T2>
{
     public abstract HashSet<T2> getPushout(functie<T1, T2> fi, HashSet<T2> codomfi,
   functie<T1, T2> gi, HashSet<T2> codomgi, HashSet<T1> domi);
}
```

2.1. Sa se scrie o clasa Pushout care extinde clasa PushoutAbstract si implementeaza metodele clasei de baza.

2.2. Sa se scrie un program care sa testeze clasa Pushout pentru exemplul de mai sus.

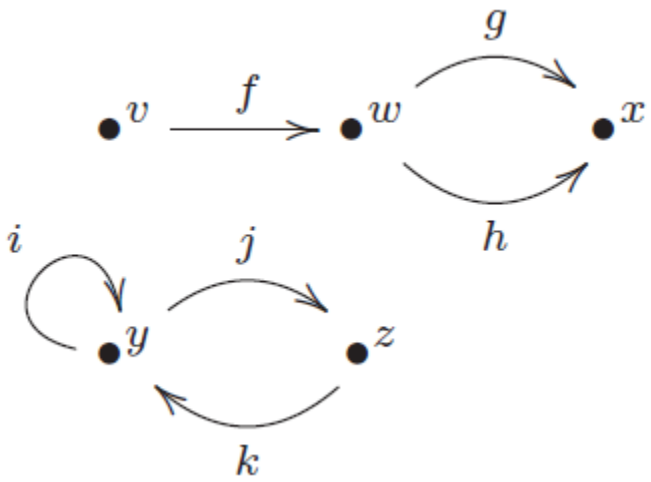
2.3. Implementati o metoda de extensie Pushout la interfata IEnumerable cu semnatura de mai jos si testati-o pe exemplu de mai sus.

```
public static class MyExtensionMethods
{
        public static HashSet<T2> Pushout<T1, T2>(this IEnumerable<T2> codomeniu,
HashSet<T1> domeniu, functie<T1, T2> fi, functie<T1, T2> gi){
```

```
        }
    }
```

3. Grafuri:

Un graf G poate fi definit ca un tuplu G=(V, A, src, tgt), unde
V este o multime, numita multimea vârfurilor lui G;
A este o multime, numita multimea arcelor lui G;
src: A $\rightarrow$ V este o funcție, numită funcția sursă care asociaza fiecarui arc din A o sursa din V;
tgt: A $\rightarrow$ V este o funcție, numită funcția tinta care asociaza fiecarui arc din A o tinta din V;.
De exemplul pentru graful G=(V, A, src, tgt) din figura:



avem V ={v, w, x, y, z} și A= {f, g, h, i, j, k}. Funcțiile sursă și țintă src, tgt: A $\rightarrow$ V sunt exprimate în tabelul următor:

A	src	tgt
f	v	w
g	w	x
h	w	x
i	y	y
j	y	z
k	z	y

3.1. Utilizand clasa Pushout implementata mai sus calculati pushout -ul functiilor src si tgt.

3.2. **Desenati grafu definit astfel:**

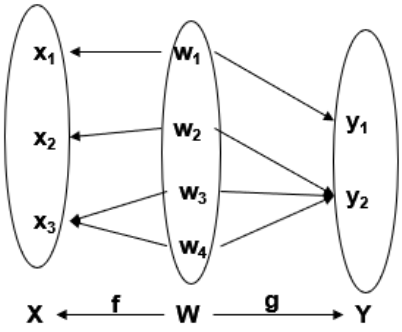
A	src	tgt
f	v	w
g	v	w
h	v	w
i	x	w
j	z	w
k	z	z

V
u
v
w
x
y
z

3.3. **Utilizand clasa Pushout implementata mai sus calculati Pushoutul functiilor src si tgt pentru acest graf.**

3.4. **Observati ca Pushoutul functiilor src si tgt in cazul grafurilor este izomorf cu multimea componentelor conexe ale grafului. Testati acest lucru adaugand componente conexe la graful de mai sus.**

Calcul Pushout - Indicatie



Relatia initiala

	x_1	x_2	x_3	y_1	y_2
x_1				1	
x_2					1
x_3					1
y_1					
y_2					

Reflexivitatea

	x_1	x_2	x_3	y_1	y_2
x_1	1			1	
x_2		1			1
x_3			1		1
y_1				1	
y_2					1

Simetria

	x_1	x_2	x_3	y_1	y_2
x_1	1			1	
x_2		1			1
x_3			1		1
y_1	1			1	
y_2		1	1		1

Relatia Reflexiva si Simetrica

	x_1	x_2	x_3	y_1	y_2
x_1	1			1	
x_2		1			1
x_3			1		1
y_1	1			1	
y_2		1	1		1

Relatia de echivalenta ρ Reflexiva, Simetrica si Tranzitiva

	x_1	x_2	x_3	y_1	y_2
x_1	1			1	
x_2		1	1		1
x_3		1	1		1
y_1	1			1	
y_2		1	1		1

Pasi de calcul Partitii

	x_1	x_2	x_3	y_1	y_2
P_1	1				
P_2	1			-1	
P_3	1			1	
P_4	1	2		1	
P_5	1	2		1	-1
P_6	1	2	-1	1	2
P_7	1	2	2	1	2

Pushout(f,g)=$\{\widehat{x_1}, \widehat{x_2}\}$

$\widehat{x_1}$=$\{x_1, y_1\}$

$\widehat{x2}$=$\{x_2, x_3, y_2\}$