

# Curs 11

Generics, Collections, Delegates, Events, Lambda  
Expressions

*Realizat de:*

*Tapuc Delia-Elena - 227*

*Vicol Marius - 227*

*Trifan Alexandra-Isabela - 227*

*Asandei Georgiana - 221*

*Toda-Puiulet Emanuela-Coralia - 227*



# Genericitatea

- **Genericitatea** este capacitatea unui limbaj de programare de a suporta definiții care pot funcționa cu diverse tipuri de date fără a specifica exact tipul în momentul definirii.
- În C#, genericitatea valabilă începând cu versiunea 2.0
- Genericele din **C#** sunt mai puternice și mai flexibile decât cele din **Java**, datorită păstrării tipurilor la runtime, suportului pentru tipuri primitive și restricțiilor mai flexibile asupra tipurilor generice.

# Metode generice

- Metodele generice permit definirea de funcționalități independente de un tip de date specific.
- Cum putem implementa functia Swap astfel incat outputul programului sa fie: x = 10, y = 5?

```
int x = 5, y = 10;  
Swap(ref x, ref y);  
Console.WriteLine($"x = {x}, y = {y}");
```

```
public static void Swap<T>(ref T a, ref T b) {  
    T temp = a;  
    a = b;  
    b = temp;  
}
```

- În C#, doar metodele și declarările de tip (cum sunt clasele, interfețele și delegațiile) pot introduce parametri generici.
- Proprietățile, indexatorii sau câmpurile trebuie să folosească parametrii generici ai clasei în care sunt definite.
- Parametrii generici pot fi introdusi în declarații de clasă, structură, interfață, delegații și metode, permitând reutilizarea codului pentru diferite tipuri de date

# Operatorul default()

- default() returnează valoarea implicită a unui tip generic T.
- Dacă T este un tip referință (cum ar fi string/class), default(T) va returna null.
- Dacă T este un tip valoare (cum ar fi int, bool, struct), default(T) va returna valoarea implicită pentru acel tip.

```
static void InitArray<T>(T[] array)
{
    for (int i = 0; i < array.Length; i++)
    {
        array[i] = default(T);
    }
}
```

```
int[] intArray = new int[3];
string[] stringArray = new string[3];

// Fiecare element devine 0
InitArray(intArray);
// Fiecare element devine null
InitArray(stringArray);
```

# Constrângerile

Constrângerile în C# pot fi aplicate atât la nivel de tip cât și la nivel de metode, iar acestea pot fi clasificate precum:

**De derivare** - indică compilatorului ca parametrul T este de tip generic, derivat dintr-un tip de baza.

**Constructor fără parametri.**

**Reference/Valoare** - T este de tip class sau struct.

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except <a href="#">Nullable</a> can be specified. See <a href="#">Using Nullable Types</a> for more information.
where T : class	The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the new( ) constraint must be specified last.
where T : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U.

# Generics and Inheritance

## ■ Independența moștenirii în generics

Dacă S este un subtip al lui T, asta nu înseamnă că `SomeGenericClass<S>` este un subtip al `SomeGenericClass<T>`. Acest concept este cunoscut ca **invariantă** în generics.

## ■ Covarianță și Contravarianță

Unele limbaje (precum C#) permit covarianță și contravarianță în generics doar în anumite situații, cum ar fi folosirea cu interfețe și delegații, dar nu și cu clase generice simple.

## ■ Implicarea în runtime

Moștenirea și legătura între tipuri generice sunt doar concepte la compilare. La runtime, toate instanțele generice sunt tratate ca același tip generic fără legătura dintre T și S.

Presupunem următoarele două clase și o clasă generică:

```
class Animal { }
class Dog : Animal { }
class Box<T> { }
```

Ce s-ar întâmpla dacă am încerca să compilăm următoarea comandă?

```
Box<Animal> animalBox = new Box<Dog>();
```

```
Type 'Dog' doesn't match the expected type 'Animal'  
Cannot convert source type 'Box<Dog>' to target type  
internal class Dog  
: Animal
```

## Nu va compila!

Dog este un subtip al clasei Animal (relația clasică de moștenire), însă Box<Dog> nu este un subtip al clasei Box<Animal>

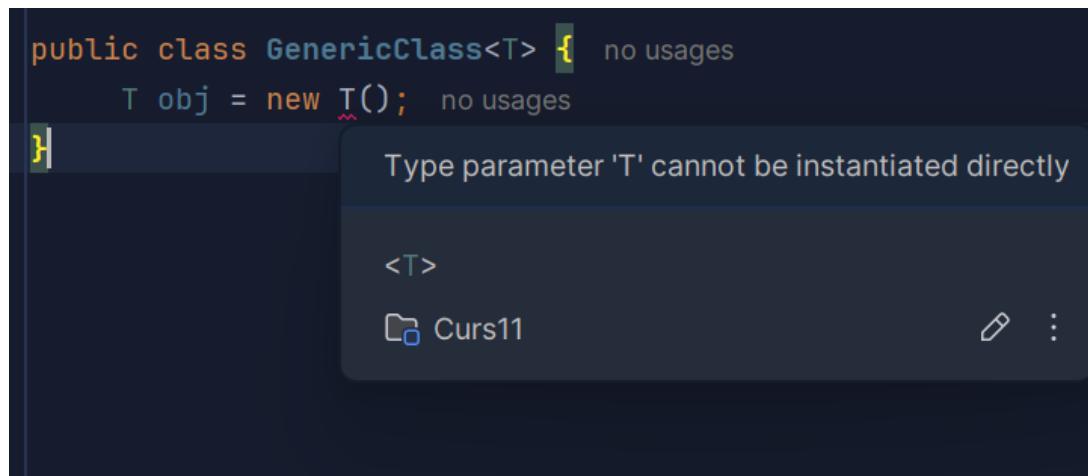
# Diferențele principale ale genericității

Java vs C#

# Type Erasure

## JAVA

- Nu există informații despre tipul generic la runtime.
- Nu poți folosi operatori specifici tipului sau instanția obiecte direct de tip generic.



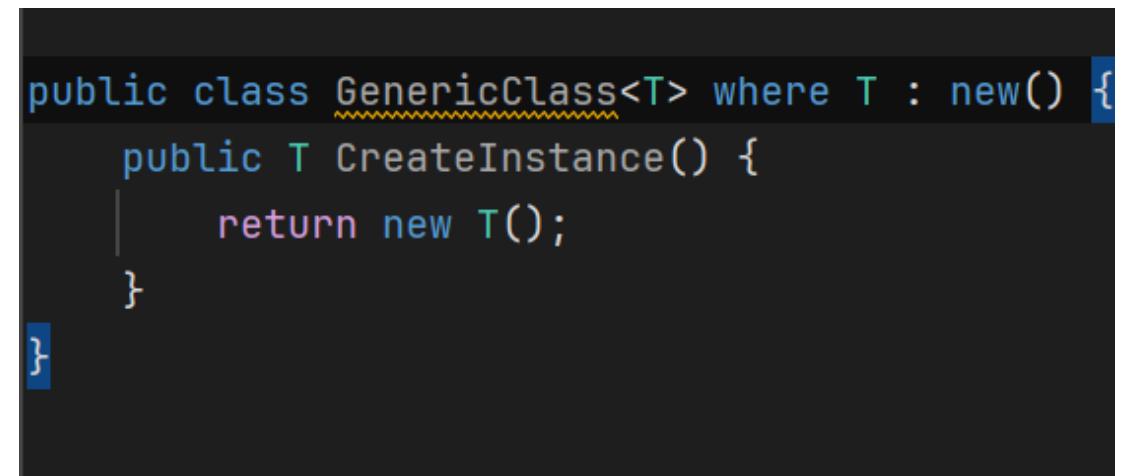
A screenshot of a Java code editor showing a generic class definition:

```
public class GenericClass<T> { no usages
    T obj = new T(); no usages
}
```

The line `T obj = new T();` is highlighted in red, and a tooltip message is displayed: "Type parameter 'T' cannot be instantiated directly".

## C#

Genericele sunt „preserveate” la runtime, ceea ce înseamnă că informațiile despre tip sunt păstrate și pot fi accesate prin reflectie.



A screenshot of a C# code editor showing a generic class definition:

```
public class GenericClass<T> where T : new() {
    public T CreateInstance() {
        return new T();
    }
}
```

# Wild Cards

## ■ Java

? extends permite folosirea unui tip derivat (Dog) acolo unde este așteptat un tip mai general (Animal)

```
class Animal {} 2 usages 1 inheritor
class Dog extends Animal {}
class Box<T> {} 2 usages

Box<? extends Animal> box = new Box<Dog>();
```

## ■ C#

out oferă aceeași flexibilitate pentru tipurile generice care sunt folosite doar ca tipuri de returnare

```
class Animal { }
  2 usages
class Dog : Animal { }
// `out` permite covariantă
  2 usages  1 inheritor
interface IBox<out T> { }

  1 usage
class Box<T> : IBox<T> { }

class MainProgram {
    static void Main()
    { // Funcționează datorită `out`
        IBox<Animal> box = new Box<Dog>();
    }
}
```

```
class Animal {} 2 usages 1 inheritor
class Dog extends Animal {} 1 usage
class Box<T> {} 2 usages

Box<? super Dog> box = new Box<Animal>();
```

În **Java**, `? super` permite folosirea unui tip mai general (`Animal`) acolo unde este așteptat un tip specific (`Dog`).

În **C#**, `in` oferă aceeași flexibilitate pentru tipurile generice care sunt folosite doar ca parametri de intrare.

```
interface IProcessor<in T> {
    void Process(T item);
}

class Processor<T> : IProcessor<T>
{
    public void Process(T item) {
        Console.WriteLine($"Processing {item.GetType().Name}");
    }

    // Funcționează datorită `in`
    public void Main() {
        IProcessor<Dog> processor = new Processor<Animal>();
    }
}
```

# Genericile primitive

- În **Java**, genericele nu pot fi utilizate cu tipuri primitive (ex. int, double), deoarece type erasure impune ca toate tipurile generice să fie convertite în obiecte.

```
public class Main {  
    public static void main(String[] args) {  
        List<int> list = new ArrayList<int>();  
        Type argument cannot be of primitive type  
        Replace 'int' with 'java.lang.Integer' Alt+Shift+Enter  
  
        List<Integer> list1 = new ArrayList<>();  
        // Corect, dar foloseste boxing/unboxing  
    }  
}
```

- În **C#** permite utilizarea tipurilor primitive cu genericile (ex. List<int>). În fundal, C# creează cod specific pentru fiecare tip utilizat, eliminând nevoia de boxing/unboxing.

```
List<int> list = new List<int>();  
list.Add(42);  
Console.WriteLine(list.Count);  
|
```

# Collections în C#

Organizarea și manipularea eficientă a datelor

# Collections C#

```
using System.Collections.Generic;  
using System.Collections;
```

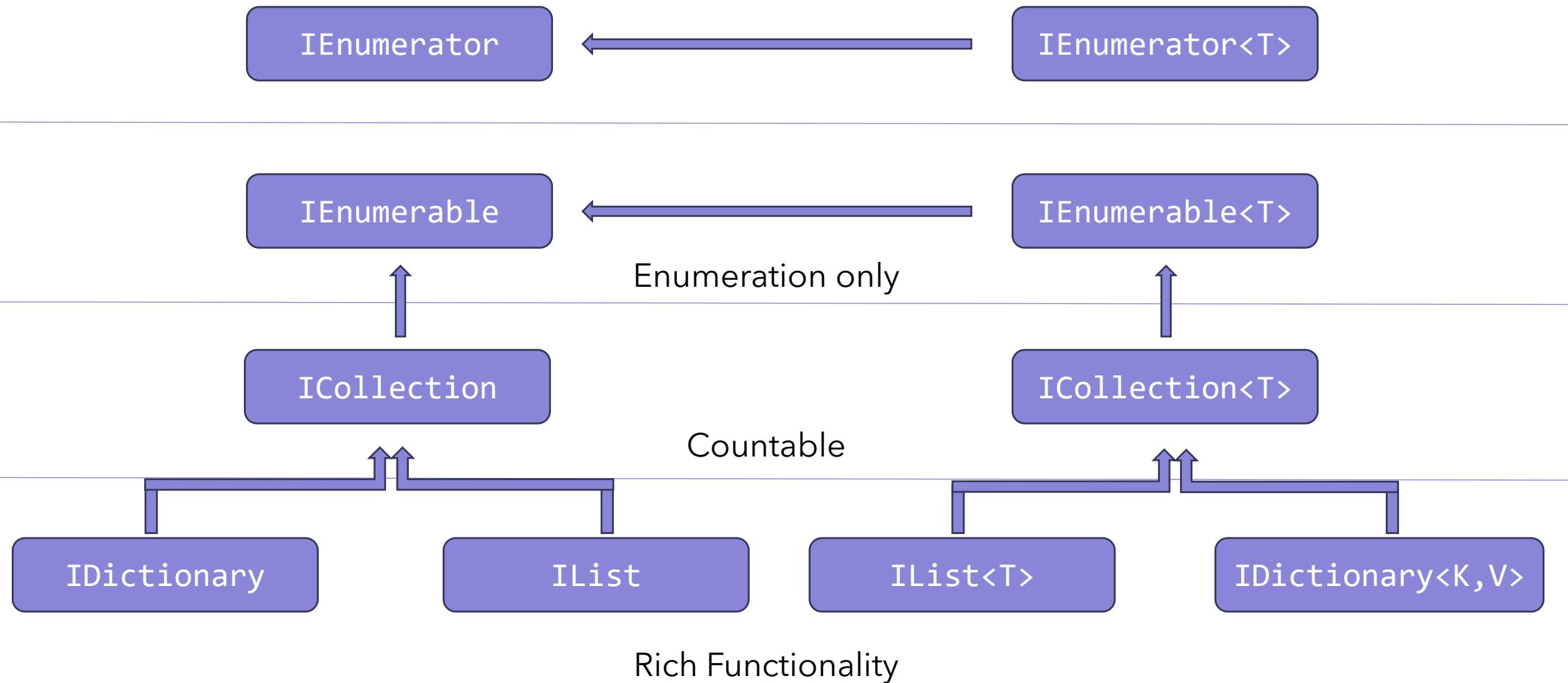
- **Colecțiiile** sunt structuri de date predefinite care ajută la stocarea, gestionarea și manipularea grupurilor de obiecte
- Namespace-urile importante pentru folosirea Colecțiilor sunt:
  - **System.Collections**;
  - **System.Collections.Generic**;

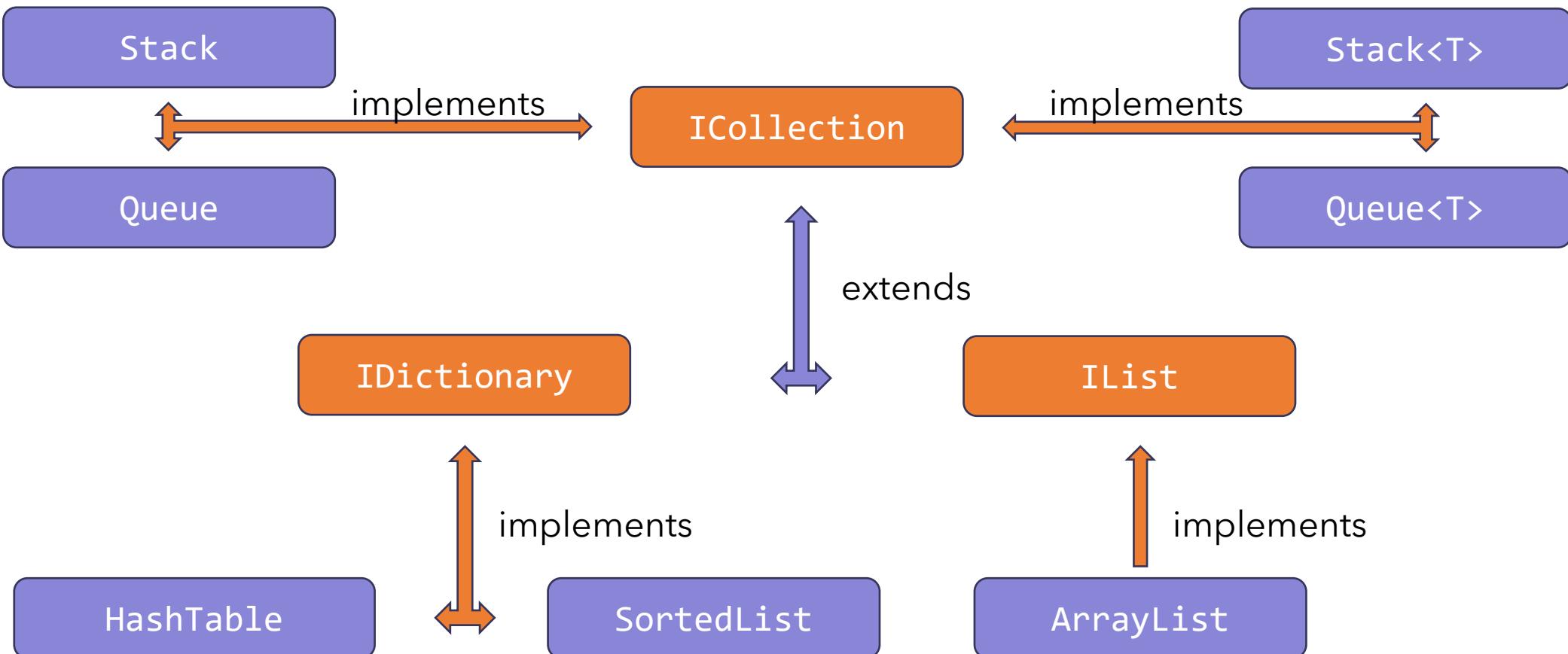
# Collections în Java VS Collections in C#

Aspect	Collections în JAVA	Collections în C#
Namespace	java.util	System.Collections, System.Collections.Generic
Framework principal	Java Collections Framework (JCF)	Generic Collections Framework
Generice	Implementate prin type erasure (informația de tip nu e disponibila la runtime)	Implementate prin reified generics (informația de tip este disponibila la runtime)
Interfețe principale	List, Set, Map	IList<T>, IDictionary<K, V>, IEnumerable<T>
Colectii imutabile	Necesită utilizarea explicită a metodelor de creare imutabilă (Collections.unmodifiableList).	Include colectii imutabile dedicate (ImmutableList<T>, ImmutableDictionary< TKey , TValue >).
Manipulare funcțională	Stream API (list.stream().filter().collect( Collectors.toList()))	LINQ (list.Where().ToList()).

## System.Collections

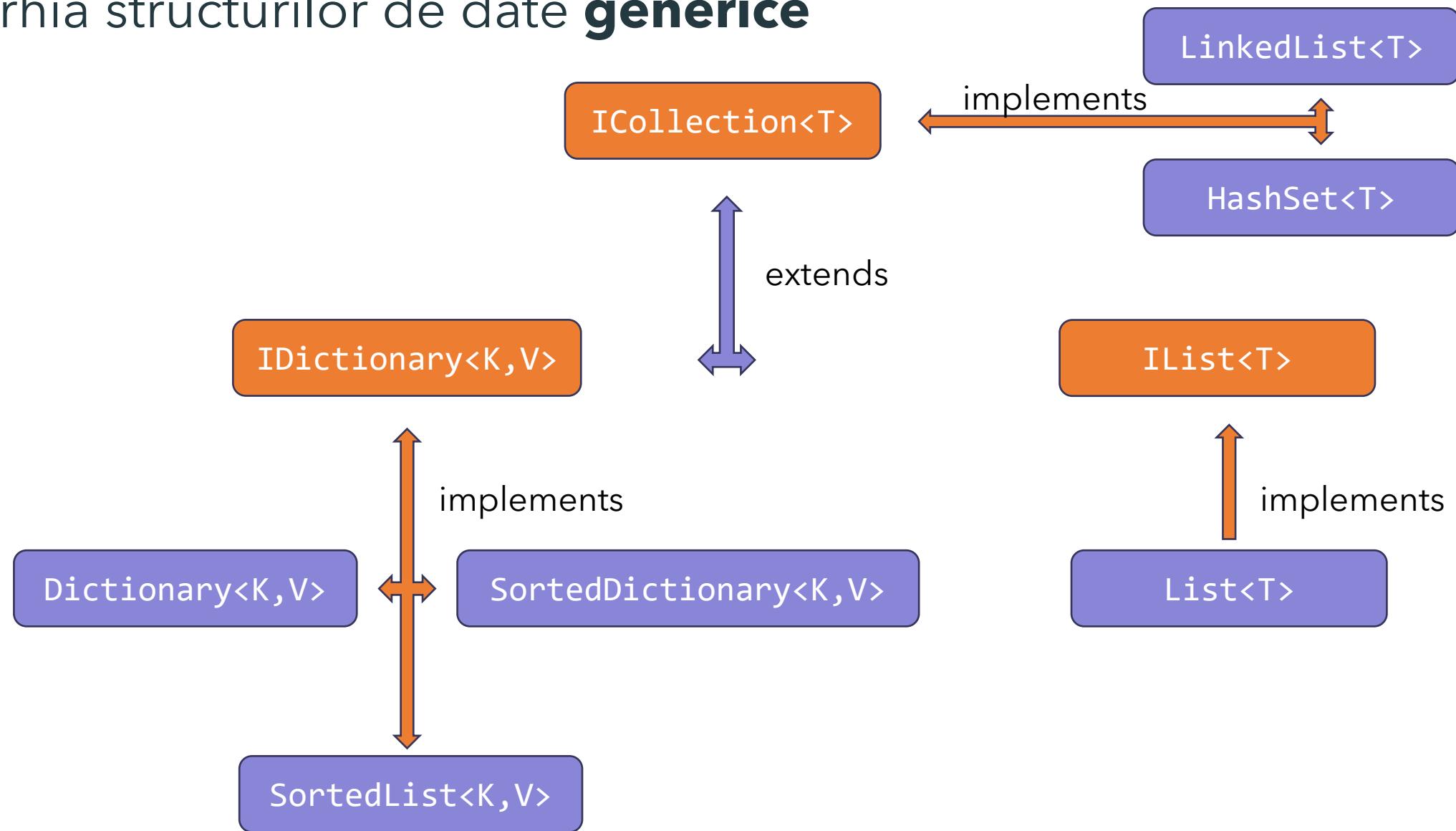
## System.Collections.Generic





Îerarhia structurilor de date **specifice**

# Ierarhia structurilor de date **generice**



# IEnumerable<T>

**IEnumerable<T>** este o *interfață generică*, care reprezintă o secvență de elemente care pot fi **enumerate**.

Este utilizată pentru a permite **iterarea** printr-o colecție de obiecte de un anumit tip, utilizând un **foreach** sau un **enumerator** (**IEnumerator<T>**).

## Exemplu:

```
IEnumerable<int> numbers = new
List<int> { 1, 2, 3, 4, 5 };
foreach (int number in numbers)
{
    Console.WriteLine("Number: " +
number);
}
```

```
#nullable enable
namespace System.Collections.Generic
{
    Exposes the enumerator, which supports a simple iteration over a collection of a specified type
    Type params: T — The type of objects to enumerate.

    ◆ IL code
    public interface IEnumerable<out T> : IEnumerable
    {
        Returns an enumerator that iterates through the collection.

        Returns: An enumerator that can be used to iterate through the collection.

        ◆ IL code
        I IEnumerator<T> GetEnumerator();
    }
}
```

# IStack<T>

Interfața **IStack<T>** este o structură de date generică ce modelează comportamentul unei stive (stack) în C#. Aceasta respectă principiul **LIFO** (Last In, First Out), unde ultimul element adăugat este primul care poate fi eliminat.

Această interfață urmând, ulterior, să fie extinsă sau implementată după bunul plac al programatorului.

```
using System.Collections.Generic;
1 usage 1 inheritor
public interface IStack<T> : IEnumerable<T>
{
    // Proprietate pentru numărul de elemente din stivă.
    1 implementation
    int Count { get; }
    // Metoda pentru a adăuga un element în stivă.
    1 implementation
    void Push(T value);
    // Metoda pentru a elimina și returna elementul din vârful stivei.
    1 implementation
    T Pop();
    // Metoda pentru a returna elementul din vârful stivei fără a-l elimina.
    1 implementation
    T Peek();
    // Metoda pentru a obține toate elementele stivei sub forma unui array.
    1 implementation
    T[] GetAllStackElements();
}
```

# IEnumerator<T>

**IEnumerator<T>** este o **interfață** care permite iterarea printr-o colecție generică, oferind acces la fiecare element din colecție în **ordine**.

Este folosită pentru implementarea Iteratorilor pe colecții custom.

**În C#, clasele interne nu au acces la membri clasei outer, din acest motiv MyIterator are o referință la Stack!**

```
// Iteator
[ 1 usage]
private class MyIterator : IEnumerator<T>
{
    private Stack<T> stack;
    private int current;
    private T[] elems;

[ 1 usage]
public MyIterator(Stack<T> stack)
{
    this.stack = stack;
    current = stack.Count - 1;
    elems = stack.GetAllStackElements();
}

public bool MoveNext()
{
    current--;
    return current >= 0;
}

public void Reset()
{
    current = stack.Count - 1;
}

[ 1 usage]
public T Current => elems[current];

object? IEnumerator.Current => Current;

public void Dispose()
{
    // Nu avem nevoie dispose in cazul acesta
}
```

# Stack<T>

Clasa generică **Stack<T>** este o implementare a interfeței **IStack<T>** folosind pentru stocarea datelor în stivă un vector generic ( **T[]** ), numărul maxim de elemente ce pot fi introduse în stivă și un index al elementului din vârful stivei.

```
// Clasa generică Stack<T> care implementează interfața IStack<T>.
[2 usages]
public class Stack<T>: IStack<T>
{
    private int _capacity;
    private T[] _elems;
    private int _top;

    // Constructorul clasei Stack, care initializează stiva cu o capacitate specificată.
    public Stack(int capacity)
    {
        this._capacity = capacity;
        _elems = new T[capacity];
        _top = -1;
    }

    // Metoda pentru obținerea unui iterator care iterează elementele stivei.
[1 usage]
    public IEnumerator<T> GetEnumerator()
    {
        return new MyIterator<T>(stack: this);
    }

    // Implementare explicită a metodei IEnumerable.GetEnumerator().
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    // Proprietate pentru numărul de elemente din stivă.
[2 usages]
    public int Count { get; }
}
```

# Problemă propusă:

Implementați în clasa generică **Stack<T>** metodele:

- void Push (T value) - adaugă în stivă un element
- T Pop() - elimină și returnează !un element! din stivă
- T Peek() - returnează !un element! din stivă fără eliminare

# DELEGATES – C#

# Delegates

- Problema: trebuie să se execute o anumită acțiune, dar nu se știe dinainte care anume, sau ce obiect va trebui efectiv utilizat.
- § Un **delegat** este un **tip referință** folosit pentru a încapsula o metoda cu un anumit antet.
- Orice metoda care are acest antet poate fi legată la un anumit delegat.

# Delegates

- => Un delegate este un tip care reprezintă referințe către metode cu o anumită listă de parametri și un anumit tip de returnare.
- Când instantiezi un delegate, poți asocia instanța sa cu orice metodă care are o semnătură (nume, tipurile și ordinea parametrilor pe care metoda îi primește) și un tip de returnare compatibile. Poți apela metoda prin instanța delegatului.
- Delegates sunt utilizati pentru a transmite metode ca argumente altor metode.

- Declarare:

```
delegate <return type> DelegateName(< list of parameters >);
```

- Exemplu:

```
delegate int ArithmeticOperation(int a, int b);
```

- Delegații definiti de utilizator sunt subclase ale clasei `System.Delegate`. Acestea sunt automat generate de compilator și nu pot fi create explicit de către utilizator.

# Exemplu

```
class Student :IComparable<Student>
{
    public int StudentID { get; set; }
    public String StudentName { get; set; }
    public int Age { get; set; }
    public override string ToString()
    {
        return StudentID + " " + StudentName + " " + Age;
    }
    public int CompareTo(Student other)
    {
        return this.StudentName.CompareTo(other.StudentName);
    }
}
```

## Problema

- Consideram entitatea Student, definita anterior.
- Definiti urmatoarele metode:
  - TeenAgerStudent
  - AdultStudent
  - RetiredStudent
  - StudentPredicate (delegate)**
- Creati o lista de student si retineti: doar studentii adulti, apoi cei pensionati, apoi adolescentii
- Folositi multicast delegate

```
delegate bool StudentPredicate(Student s);
```

```
public static bool TeenAgerStudent(Student s)
{
    return s.Age > 12 && s.Age < 20;
}
```

```
public static bool AdultStudent(Student s)
{
    return s.Age >= 20;
}
```

```
Student s = new Student() { StudentID = 1, StudentName = "Cris", Age = 19 };

StudentPredicate testIfTeenAger = new StudentPredicate(TeenAgerStudent);
Console.WriteLine(testIfTeenAger(s));

StudentPredicate testIfAdult = new StudentPredicate(AdultStudent);
Console.WriteLine(testIfAdult(s));

//metode anonime
StudentPredicate testIfRetired = delegate (Student stud) {return stud.Age > 60;};
Console.WriteLine(testIfRetired(s));

//lambda
StudentPredicate testIfChild = stud => stud.Age <=12 ;
Console.WriteLine(testIfChild(s));
```

# Metodele anonoime în C#

- Metodele anonoime sunt o funcționalitate în C# care permite definirea unei metode inline fără a-i atribui explicit un nume. Ele sunt utilizate de obicei împreună cu delegate pentru a simplifica codul și pentru a evita necesitatea de a defini metode separate.
- Metodele anonoime au fost introduse în **C# 2.0**, iar în versiunile ulterioare (în special C# 3.0), au fost înlocuite parțial de expresiile **lambda**, care oferă o sintaxă și mai concisă.

## Sintaxa metodei anonte

```
delegate <tip_return> NumeDelegate(<parametri>);  
NumeDelegate variabila = delegate(<parametri>)  
{  
    // Codul metodei anonte  
};
```

# Multicast Delegates

```
StudentPredicate testStudent = AdultStudent;
// sau new StudentPredicate(AdultStudent);
Console.WriteLine(testStudent(s));

//multicast delegate
testStudent += TeenAgerStudent;
Console.WriteLine(testStudent(s));
```

- Operatorul += adaugă metoda TeenAgerStudent la lista de metode pe care delegate-ul testStudent le va apela.
- Acum, testStudent devine un **multicast delegate**, ceea ce înseamnă că face referire la mai multe metode (în ordinea în care au fost adăugate).
- Când un **multicast delegate** este apelat, toate metodele din lista sa sunt apelate, în ordine.
- Dacă delegate-ul are o valoare de returnare (în cazul nostru, bool), doar valoarea **returnată de ultima metodă din lanț** va fi luată în considerare și afișată de Console.WriteLine.

# Generics and Delegates - filter students

```
delegate bool Predicate<T>(T entity);
```

//Un delegate generic, care reprezintă o metodă ce primește un parametru de tipul T și returnează un bool.

```
public List<T> Filter<T>(List<T> list, Predicate<T> test)
```

```
{
```

```
    List<T> res = new List<T>();
```

```
    foreach (var entity in list)
```

```
        if (test(entity))
```

```
            res.Add(entity);
```

```
    res.Sort();
```

```
    return res;
```

```
}
```

//Aceasta este o metodă generică pentru filtrarea unei liste pe baza unui criteriu dat de delegate-ul Predicate<T>.

```
public List<Student> filterTeenAgerStudents()
{
    return Filter(studentList, TeenAgerStudent);
}

studentList=InitStudentList();

List<Student> list=filterTeenAgerStudents();

Console.WriteLine("TeenAger Students Ascending by Name");
foreach (var s in list)
    Console.WriteLine(s);
```

# Generics and Delegates - Filter&Sorter

```
delegate bool Predicate<T>(T entity);

public List<T> FilterAndSorter<T>(List<T> list, Predicate<T> test, Comparison<T> comp)
{
    List<T> res = Filter(list, test);
    res.Sort(comp);
    return res;
}
```

În C#, `Comparison<T>` este un tip delegat predefinit în biblioteca standard. Acceptă două argumente de tipul `T` și returnează un întreg (`int`). Semnificația valorii returnate este:

- < 0: Primul obiect (`x`) este considerat mai mic decât al doilea obiect (`y`).
- = 0: Cele două obiecte sunt egale.
- > 0: Primul obiect (`x`) este considerat mai mare decât al doilea obiect (`y`).

```
public List<Student> FilterTeenAgerStudentsDescByAge()
{
    return FilterAndSorter(studentList, TeenAgerStudent, (x, y) => { return -(x.Age - y.Age); });
}

studentList=InitStudentList();

list = FilterTeenAgerStudentsDescByAge();

Console.WriteLine("TeenAger Students Descending by Age");
foreach (var s in list)
    Console.WriteLine(s);
```

EVENIMENTE

# Evenimente

Implementează formal modelul *Publisher/Subscriber* (*Observer*)

**Publisher:** Obiectul care declară și lansează evenimentul.

**Subscriber (Observer):** Obiectul care se abonează la eveniment și gestionează apariția lui.

Este un **wrapper pentru un delegate**. În esență:

- Expune doar o parte a funcționalităților unui delegate, astfel încât să fie utilizate doar acele aspecte necesare pentru modelul Publisher/Subscriber.
- Este o metodă care poate fi apelată automat de mai mulți subscribers atunci când un anumit eveniment are loc. Un subscriber poate gestiona mai multe evenimente de la mai mulți publishers.
- Previne accesul direct la delegate, prin urmare și interferența între subscribers.
- Când un eveniment are mai mulți subscribers, handler-ele evenimentelor sunt invocate sincron atunci când un eveniment este declanșat.
- Evenimentele care nu au subscribers nu sunt niciodată declanșate.
- Evenimentele sunt utilizate în general pentru a semnala acțiuni ale utilizatorului cum ar fi apăsarea de butoane sau selecții în meniu în cadrul interfețelor grafice. Reprezintă un mecanism sigur de notificare.

## 1. Definirea unei metode delegate publică

```
public delegate void DelegateEvent(Object sender, EventArgs args);
```

- Semnătura delegate-ului se compune din 2 parametri:
  - **sender**: sursa evenimentului, el îl lansează
  - **args**: păstrează anumite informații despre eveniment, transmise consumatorului de eveniment (event handler) sau metodei care tratează evenimentul

## 2.Definirea unei clase care lansează evenimentul (Publisher)

```
class Publisher
{
    public event DelegateEvent eventName;
    ...
    someMethod(...)
    {
        EventArgsSubClass args = new EventArgsSubClass(some data);

        //cod care declanșează evenimentul
eventName(this, args);      //==  eventName?.Invoke(this, args)
        //sau eventName(this, null);
    }
}
```

### 3.Definirea unei clase care tratează apariția evenimentului (Observer)

```
class Observer
{
    //metodele care se potrivesc cu semnătura delegate-ului
    public void OnEventName(Object sender, EventArgs args)
    {
        //cod pentru gestionarea evenimentului (event handling code)
    }
    ...
}

//crearea unui Publisher
Publisher pub = new Publisher(...);
//crearea de Observers
Observer obs1 = new Observer(...);
Observer obs2 = new Observer(...);
//abonarea Observers la eveniment
pub.eventName += new DelegateEvent(obs1.OnEventName);
pub.eventName += new DelegateEvent(obs2.OnEventName);
pub.someMethod(...); //apel explicit al metodei care declanșează evenimentul
```

### 4.Crearea unui Publisher și a unor Observers

## 4. Implementare

```
public class EventArgsSubClass : EventArgs
{
    □ 2 usages
    public string Data { get; }

    □ 1 usage
    public EventArgsSubClass(string data)
    {
        Data = data;
    }
}

public delegate void DelegateEvent(Object sender, EventArgs args);

□ 2 usages
public class Publisher
{
    public event DelegateEvent eventName;
    □ 1 usage
    public void SomeMethod()
    {
        EventArgsSubClass args = new EventArgsSubClass(data: "Some data");
        if (eventName != null)
        {
            eventName(sender: this, args);
        }
    }
}
```

```
public class Observer
{
    □ 2 usages
    public void OnEventName(Object sender, EventArgs args)
    {
        if (args is EventArgsSubClass customArgs)
        {
            Console.WriteLine($"Eveniment gestionat! Sursa: {sender}, Date: {customArgs.Data}");
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Publisher pub = new Publisher();

        Observer obs1 = new Observer();
        Observer obs2 = new Observer();

        pub.eventName += new DelegateEvent(obs1.OnEventName);
        pub.eventName += new DelegateEvent(obs2.OnEventName);

        pub.SomeMethod();
    }
}
```

## În cazul în care...

- se declanșează un eveniment fără parametrii => parametru specific EventArgs.Empty

```
MyEvent?.Invoke(this, EventArgs.Empty);
```

- se declanșează un eveniment cu parametrii personalizați => se poate utiliza genericitatea

```
public delegate void EventHandler<TMyEventArgs>(Object sender, EventArgs args)
    where TMyEventArgs : MyEventArgs;

public class Publisher
{
    public event EventHandler<MyEventArgs> MyEvent;
    (...)
```

# C# vs. Java

```
public class Publisher
{
    // Definirea unui eveniment bazat pe delegate-ul EventHandler
    public event EventHandler MyEvent;

    public void RaiseEvent()
    {
        Console.WriteLine("Eveniment declanșat în Publisher.");
        MyEvent?.Invoke(this, EventArgs.Empty); // Lansează evenimentul
    }
}

public class Observer
{
    public void HandleEvent(object sender, EventArgs args)
    {
        Console.WriteLine("Eveniment capturat în Observer.");
    }
}

public class Program
{
    public static void Main()
    {
        Publisher publisher = new Publisher();
        Observer observer = new Observer();

        // Abonarea Observer-ului la evenimentul din Publisher
        publisher.MyEvent += observer.HandleEvent;

        // Declansarea evenimentului
        publisher.RaiseEvent();
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class Publisher { 2 usages
    private List<EventListener> listeners = new ArrayList<>(); 2 usages

    // Metodă pentru a adăuga un Subscriber (Observer)
    public void addListener(EventListener listener) { listeners.add(listener); }

    // Declansarea evenimentului
    public void raiseEvent() { 1 usage
        System.out.println("Eveniment declansat in Publisher.");
        for (EventListener listener : listeners) {
            listener.onEvent(); // Notificarea Subscribers
        }
    }
}

public interface EventListener { 4 usages 1 implementation
    void onEvent(); 1 usage 1 implementation
}

public class Observer implements EventListener { 2 usages
    @Override 1 usage
    public void onEvent() { System.out.println("Eveniment capturat in Observer."); }
}

public class Main {
    public static void main(String[] args) {
        Publisher publisher = new Publisher();
        Observer observer = new Observer();

        // Abonarea Observer-ului la evenimentul din Publisher
        publisher.addListener(observer);

        // Declansarea evenimentului
        publisher.raiseEvent();
    }
}
```

Aspect	C#	Java
Mecanism integrat	Are suport nativ pentru events cu event și delegate.	Nu există suport nativ; trebuie să se folosească interfețe și liste personalizate.
Definirea unui eveniment	Prin utilizarea cuvântului-cheie event în combinație cu delegates (EventHandler).	Se creează o interfață și se implementează manual lista de listeners.
Abonarea la evenimente	Folosește operatorul += pentru a lega un Observer la un eveniment.	Adaugă Observer-ul la o listă folosind o metodă personalizată (addListener).
Declanșarea evenimentelor	Se face folosind metoda .Invoke() sau operatorul ?.	Se iterează prin lista de Observers și se apelează metoda corespunzătoare.
Siguranță	Doar Publisher-ul poate declanșa evenimentele.	Oricine poate apela metoda raiseEvent.

## Exercițiu

- Vom crea o aplicație simplă care folosește un event pentru a notifica un subscriber atunci când un temporizator ajunge la 0.
- Scenariu: Un temporizator (Timer) numără invers de la un număr dat. Când ajunge la 0, declanșează un eveniment care notifică o clasă subscriber că temporizatorul s-a terminat.

Pe scurt:

- Evenimentele sunt o modalitate formală de a implementa modelul Publisher/Subscriber.
- Ele sunt bazate pe *delegates*, care definesc semnătura metodelor ce pot gestiona evenimentul.
- Rolul lor principal este de a preveni manipularea necontrolată a *delegates* și de a asigura un mecanism sigur de notificare între componente.
- Prin folosirea evenimentelor, codul devine mai modular și mai ușor de extins.

# Lambda expressions

- Folosite în mod asemănător cu funcțiile anonime pentru a crea o funcție inline

# Lambda expressions

- Folosite în mod asemănător cu funcțiile anonime pentru a crea o funcție inline (nu e necesar specificarea tipului inputului)

# Lambda expressions

- Există două tipuri de Expresii Lambda:

# Lambda expressions

- Există două tipuri de Expresii Lambda:
  1. Expression Lambda

# Lambda expressions

- Există două tipuri de Expresii Lambda:
  1. Expression Lambda
  2. Statement Lambda

# Lambda Expressions

	<b>Expression Lambda</b>	<b>Statement Lambda</b>
Sintaxa	<code>input =&gt; expression;</code>	<code>input =&gt; { statements };</code>

# Lambda Expressions

	<b>Expression Lambda</b>	<b>Statement Lambda</b>
Sintaxa	<code>input =&gt; expression;</code>	<code>input =&gt; { statements };</code>
Returnarea valorii	Se returnează implicit	Necesită utilizarea explicită return

# Lambda Expressions

```
Func<int, int> square = x => x * x;  
Console.WriteLine(square(5));
```

# Lambda Expressions

```
Action<int> printSquare = x =>
{
    int result = x * x;
    Console.WriteLine(result);
};

printSquare(5);
```

Mulțumim pentru atenție!