

Platforma Microsoft .NET definește o gamă largă de colecții pentru păstrarea obiectelor. Colecțiile sunt mai flexibile decât vectorii, deoarece dimensiunea acestora poate fi modificată și, de asemenea, există o gamă largă de metode pentru adăugarea, ștergerea și accesarea elementelor unei colecții.

Overview


- Colecții orientate obiect
- Colecții generice
- Colecții specializate



Acest modul introduce colecțiile orientate pe obiecte ce se găsesc în **System.Collections**. Se descriu apoi colecțiile generice aflate în Namespace-ul **System.Collections.Generic**, ce au fost introduse în .NET 2.0. Acest modul descrie diferențele dintre colecțiile bazate pe obiecte și colecțiile generice. Acest modul prezintă de asemenea și câteva din colecțiile specializate aflate în Namespace-ul **System.Collections.Specialized**.


La finalul acestui modul, veți putea să:

- Folosiți colecții orientate pe obiecte ce se găsesc în Namespace-ul **System.Collection**
- Folosiți colecții generice ce sunt definite în Namespace-ul **System.Collections.Generic**
- Folosiți colecții specializate ce sunt definite în Namespace-ul **System.Collections.Specialized**



Colecții orientate pe obiecte

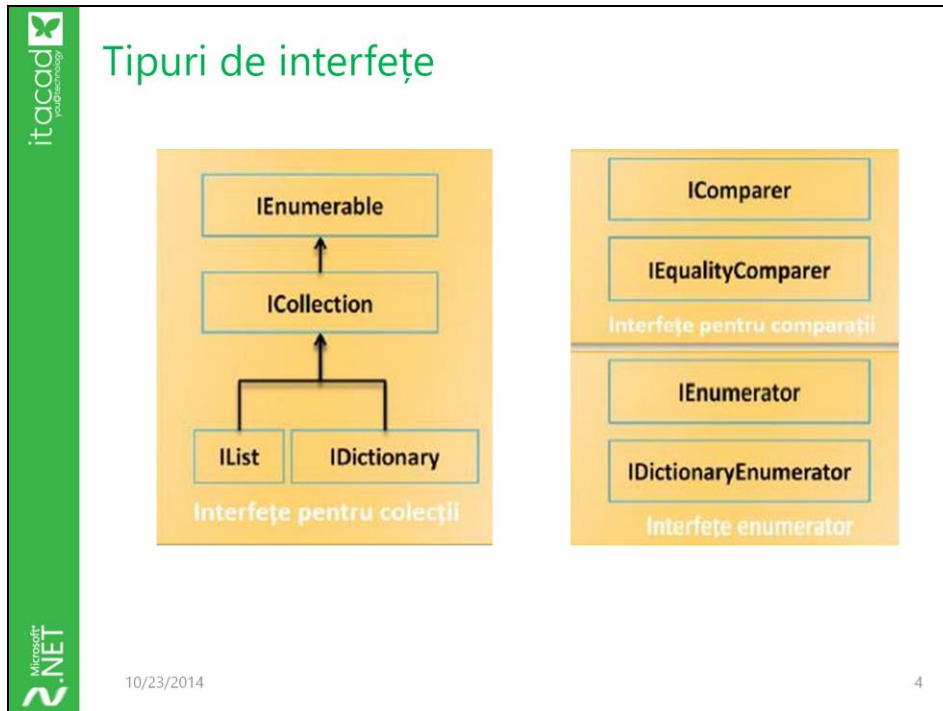
- ▶ Interfețe pentru colecții
- ▶ Folosirea colecțiilor **ArrayList**
- ▶ Folosirea colecțiilor **Queue** și **Stack**
- ▶ Folosirea colecțiilor **Hashtable** și **SortedList**
- ▶ Folosirea clasei **BitArray**
- ▶ Implementarea claselor colecții puternic tipate



10/23/2014 3



Clasele orientate pe obiecte sunt definite în namespace-ul **System.Collections**. Acest namespace definește de asemenea un set de interfețe ce specifică operații comune ce sunt implementate de aceste clase colecție. Puteți folosi aceste colecții în orice aplicație bazată pe platforma .NET

Această lecție descrie clasele și interfețele din namespace-ul **System.Collections** și oferă exemple de utilizare pentru fiecare dintre acestea.



Namespace-ul System.Collections definește interfețe ce specifică comportamentul pentru majoritatea colecțiilor din platforma .NET. Aceste interfețe pot fi organizate în 3 grupuri, cum urmează:

- interfețe pentru colecții
- interfețe pentru comparații
- interfețe enumerator



- ▶ **Interfețe pentru colecții**
 - ▶ IEnumerable
 - ▶ GetEnumerator()
 - ▶ ICollection
 - ▶ CopyTo()
 - ▶ IList
- ▶ **Interfețe pentru comparații**
 - ▶ IComparer
 - ▶ Compare()
 - ▶ IEqualityComparer
 - ▶ Equals()
 - ▶ GetHashCode()

10/23/2014

5

Interfețele pentru colecții:

- IEnumerable

Definește o metodă **GetEnumerator** ce întoarce un obiect care implementează interfața **IEnumerator**. Acest enumerator este folosit pentru a itera pe elementele unei colecții.

- ICollection

Derivată din interfața **IEnumerable**, definește dimensiunea, enumeratori și membri de sincronizare ce pot fi folosiți cu orice colecție de obiecte.

Interfața **ICollection** definește metoda **CopyTo** ce copiază elementele din **ICollection** într-un vector, începând de la un anumit index.

- IList

Derivată din interfața **ICollection** și reprezintă o colecție de obiecte ce pot fi accesate individual folosind un index.

- IDictionary

Această interfață este derivată din interfața **ICollection** și reprezintă o colecție de perechi cheie/valoare.

Interfețele pentru comparații:

- IComparer



Definește o metodă **Compare** ce primește ca parametrii două obiecte și întoarce o valoare ce indică dacă primul obiect este mai mic, egal sau mai mare decât al doilea obiect.

Implementarea decide ce înseamnă ca un obiect să fie mai mare sau mai mic decât altul.

Mai multe clase colecție oferă un constructor ce primește ca parametru un obiect de tip **IComparer**. Acesta permite specificarea cărei implementări ale **IComparer** să fie folosită de colecție. Implementarea implicită este clasa **Comparer**, ce execută comparații case-sensitive pe stringuri. Dacă se doresc comparații ce nu sunt case-sensitive, folosiți clasa **CasInsensitiveComparer**.

- IEqualityComparer

Definește o metodă **Equals** ce determină dacă obiectele specificate sunt egale. Interfața definește de asemenea o metodă **GetHashCode** ce întoarce un hashcode pentru obiectul specificat.



► **Interfețe pentru comparații între obiecte**

- **IComparable**
 - CompareTo()

► **Interfețe enumerator**

- **IEnumerator**
 - MoveNext()
- **IDictionaryEnumerator**
 - Enter, Key, Value

10/23/2014 6

Namespace-ul **System** definește o interfață pentru obiecte în general ce permite unui obiect să se compare cu un altul, după cum urmează:

- **IComparable**

Definește o metodă **CompareTo** ce compară obiectul curent cu un altul și întoarce o valoare ce indică dacă obiectul curent este mai mic, egal sau mai mare ca obiectul primit ca parametrul. Interfața **IComparable** este implementată de acele timpuri de date ce pot fi ordonate, cum ar fi tipurile numerice sau stringurile.

Interfețe enumerator

Namespace-ul **System.Collections** definește două interfețe enumerator, după cum urmează:



- **IEnumerator**

Suportă enumerații pentru o colecție de obiecte. Un obiect de tipul **IEnumerator** este obținut când este invocată metoda **GetEnumerator** a interfeței **ICollection**. **IEnumerator** definește o proprietate **Current** ce indică itemul curent din colecție, o metodă **MoveNext** ce mută enumeratorul către următorul item din colecție și o metodă **Reset** ce setează enumeratorul la poziția inițială.

- **IDictionaryEnumerator**

Moștenește interfața **IEnumerator** și suportă enumerații peste un dicționar de obiecte. Un obiect de tipul **IDictionaryEnumerator** se obține la invocarea metodei **GetEnumerator** a unui obiect de tipul **IDictionary**.

Pe lângă membrii ce sunt definiți de către interfața de bază, **IDictionaryEnumerator** definește proprietățile **Enter**, **Key** și **Value**, ce permit accesul la intrarea curentă în dicționar și la valoarea asociată.

Folosirea colecțiilor **ArrayList**

- ▶ **ArrayList** este o colecție ce păstrează ordinea elementelor
- ▶ Elementele pot fi accesate folosind un index *zero-based*
- ▶ Dimensiunea se modifică automat la adăugarea elementelor

Exemplu:

```
ArrayList al = new ArrayList();
al.Add("Paris");
al.Add("Ottowa");
...
object obj = al[1];
```

10/23/2014
7

Clasa **ArrayList** implementează interfețele **ICollection**, **ICollection**, **ICollection** și **ICollection**. Capacitatea unui **ArrayList** este numărul de elemente pe care le poate reține. La adăugarea unui obiect într-un **ArrayList**, obiectul **ArrayList** își modifică automat capacitatea. Pentru a reduce capacitatea unui obiect **ArrayList**, apălați metoda **TrimToSize**.

Clasa **ArrayList** implementează toate metodele și proprietățile interfețelor **ICollection**, **ICollection**, **ICollection**, **ICollection**. **ArrayList** are de asemenea o proprietate **Capacity** ce indică ce capacitate curentă are obiectul **ArrayList**.

Următorul exemplu de cod arată cum se crează și folosește un obiect de tipul **ArrayList**. Exemplul arată de asemenea cum se iterează prin elementele obiectului **ArrayList** folosind un enumerator și o instrucțiune de ciclare **foreach**.

```
using System.Collections;

// Instanțierea și adăugarea de obiecte
ArrayList al = new ArrayList();
al.Add("Paris");
al.Add("Ottowa");
al.AddRange(new string[] { "Rome", "Tokyo", "Tunis",
    "Canberra" });

// Obținerea obiectelor 'Paris', 'Ottowa', și 'Rome'.
ArrayList firstThree = al.GetRange(0, 3);



// Ștergerea obiectelor 'Tokyo' și 'Tunis'.
al.RemoveRange(3, 2);

// Accesarea, inserarea și ștergerea unor obiecte ce se află la
// anumite locații
```

```
object itemOne = al[1]; // Accesarea 'Ottowa'.
al.Insert(1, "Moscow"); // Mutarea la dreapta a
'Ottowa' și a celorlalte elemente
al.RemoveAt(2); // Ștergerea 'Ottowa'.

// Sortarea elementelor vectorului și căutarea
obiectului "Moscow"
al.Sort();
int indexOfMoscow = al.BinarySearch("Moscow");

// Iterarea prin ArrayList. Rezultatul afișării este:
'Canbera //Moscow Paris Rome'.
foreach (object obj in al)
{
    Console.WriteLine("{0} ", obj);
}
```

Folosirea colecțiilor **Queue** și **Stack**

- ▶ **Queue**: FIFO -first in, first out
- ▶ **Stack**: LIFO -last in, first out

Exemplu:

```
// Folosirea Queue.
Queue aQueue = new Queue();
aQueue.Enqueue("Item 1");
aQueue.Enqueue("Item 2");
object obj = aQueue.Dequeue();

// Folosirea Stack.
Stack aStack = new Stack();
aStack.Push("Item 1");
aStack.Push("Item 2");
obj = aStack.Pop();
```

10/23/2014
8

Clasa **Queue** reprezintă o colecție de obiecte FIFO. Primul obiect introdus va fi primul obiect ce va fi extras.

Clasa **Stack** reprezintă o colecție LIFO. Ultimul obiect introdus va fi primul obiect extras.

Queue și **Stack** implementează interfețele **ICollection**, **IEnumerable** și **ICloneable**.

Folosirea clasei **Queue**

Clasa **Queue** implementează coada ca un vector circular în care obiectele se inserează printr-un capăt și sunt extrase prin celălalt. Capacitatea unui **Queue** este numărul de obiecte ce poate fi reținut în **Queue**. La adăugarea unor elemente în **Queue**, aceasta se va redimensiona automat prin realocare. Pentru a reduce capacitatea unui obiect **Queue**, se apelează metoda **TrimToSize**.

Pe lângă metodele și proprietățile interfețelor **ICollection**, **IEnumerable** și **ICloneable**, clasa **Queue** mai oferă acces la o serie de alte metode:

aQueue.Peek() - obținerea obiectului 'Item 1', fără a îl extrage din coadă

aQueue.Dequeue() - extragerea din coadă a primului element

aQueue.TrimToSize() - redimensionarea cozii



Folosirea clasei **Stack**

Clasa **Stack** implementează o stivă folosind un vector circular în care elementele sunt inserate printr-un capăt și extrase tot de acolo. Capacitatea unui obiect **Stack** este numărul de elemente pe care obiectul le poate stoca. La adăugarea de elemente, capacitatea este mărită automat prin realocare.

Stack implementează toate metodele și proprietățile din **ICollection**, **IEnumerable** și **ICloneable**. Metode:

aStack.Peek() - obținerea elementului 'Item 4' fără a îl extrage

aStack.Pop() - extragerea 'Item 4'

Folosirea colecțiilor **Hashtable** și **SortedList**

- ▶ **Hashtable**: dicționar de perechi cheie/valoare
- ▶ **SortedList**: dicționar sortat de perechi cheie/valoare

Exemplu:

```
// Using Hashtable.
Hashtable t = new Hashtable();
t.Add("Chai", 18.0);
t.Add("Chang", 22.5);
object obj = t["Chai"];

// Using SortedList.
SortedList s = new SortedList();
s.Add("Chai", 18.0);
s.Add("Chang", 22.5);
obj = s["Chai"]; // obj = s[0];
obj = s.GetByIndex(1);
```

10/23/2014
9

Clasa **Hashtable** reprezintă un dicționar de perechi cheie/valoare ce sunt organizate în funcție de codul hash al cheii.

Clasa **SortedList** reprezintă o colecție de perechi chei/valoare ce sunt sortate în funcție de chei și sunt accesate fie prin cheie, fie prin index.

Utilizarea clasei **Hashtable**

Clasa **Hashtable** implementează interfețele **IDictionary**, **ICollection**, **IEnumerable**, **ISerializable**, **IDeserializationCallback** și **ICloneable**. Fiecare intrare într-un obiect **Hashtable** este stocat ca un obiect **DictionaryEntry**. Cheile dintr-un obiect **Hashtable** trebuie să suprascrie metoda **Object.GetHashCode** și metoda **Object.Equals**.

Clasa **Hashtable** implementează toate metodele și proprietățile definite în interfețele de bază. Metode: **Contains()**, **Remove()**, iterare cu **foreach**.


Utilizarea clasei **SortedList**

Clasa **SortedList** implementează interfețele **IDictionary**, **ICollection**, **IEnumerable** și **ICloneable**. Elementele dintr-un **SortedList** pot fi accesate fie prin cheie, fie prin index. Accesarea prin cheie se face asemănător unui **IDictionary**, accesarea prin index se face asemănător unui **ICollection**.

La adăugarea unui element, el este inserat în **SortedList** în ordinea corectă și indecșii sunt ajustați în concordanță. La scoaterea unui element, de asemenea indexarea este ajustată. De aceea, indexul unei perechi cheie/valoare se poate schimba pe măsură ce elemente sunt adăugate și eliminate.

Clasa **SortedList** implementează toate metodele și proprietățile ce sunt definite în clasele interfață de bază. Există de asemenea o proprietate numită **Capacity** în care este reținută capacitatea curentă a obiectului.

Metode: **IndexOfKey()**, **GetByIndex()**, **Remove()**, **RemoveAt()**, iterare cu **foreach**.



Folosirea clasei **BitArray**


- ▶ **BitArray**
 - ▶ Un vector de flag-uri binare ce conține valori *Booleane*
 - ▶ **BitArray** definește metode ce execută operații orientate pe biți

Exemplu:

```
BitArray ba1 = new BitArray(3);
ba1[0] = true;
ba1[1] = false;
ba1[2] = true;

BitArray ba2 = ... ;

ba1.And(ba2);
ba1.Or(ba2);
ba1.Xor(ba2);
```



10/23/2014 10

Clasa **BitArray** conține un vector de măști pentru biți. Fiecare astfel de mască reprezintă o valoare Booleană. Folosiți **BitArray** dacă aveți mai multe valori Booleane pe care doriți să le stocați eficient.



Folosirea clasei **BitArray**

Clasa **BitArray** implementează interfețele **ICollection**, **IEnumerable** și **ICloneable**. La crearea unui obiect **BitArray**, specificați numărul de biți pe care doriți să îl rețineți. Puteți accesa un bit în obiectul **BitArray** prin poziția sa, indexată de la 0.

Folosire:

```
// Crearea unui BitArray și inițializarea imediată a biților.
BitArray ba2 = new BitArray(new bool[] { true, true, false });
// Setarea bitilor
ba1[2] = true; // ba1: false, false, true.
ba2.SetAll(true); // ba2: true, true, true.

// Iterarea prin elementele din BitArray
foreach (bool bit in ba1)
{
    Console.WriteLine("{0} ", bit);
}
```



Implementarea claselor colecții puternic tipate

- ▶ Se pot defini clase colecții noi extinzând următoarele clase:
 - ▶ **CollectionBase**
 - ▶ OnClear, OnInsert ...
 - ▶ **ReadOnlyCollectionBase**
 - ▶ **DictionaryBase**
- ▶ În cele mai multe cazuri, se vor folosi colecții generice în loc de a defini colecții proprii

10/23/2014 11

Namespace-ul **System.Collections** definește următoarele clase de bază abstracte ce pot fi extinse pentru a implementa colecții puternic tipate:

- **CollectionBase**
- **ReadOnlyCollectionBase**
- **DirectoryBase**

Implementarea unei colecții folosind **CollectionBase**

Clasa **CollectionBase** implementează interfețele **ICollection** și **IEnumerable** și oferă funcționalitate de bază pe care o puteți extinde pentru a implementa o clasă colecție proprie, modificabilă, puternic tipată.

CollectionBase definește o proprietate publică numită **Capacity** ce setează sau citește numărul de elemente pe care colecția le poate conține și o proprietate **Count** ce citește numărul de elemente ce sunt în colecție la un moment. **CollectionBase** definește de asemenea o proprietate protected **InnerList** ce întoarce într-un obiect **ArrayList** elementele din colecție. Totodată, mai definește proprietatea **List** ce întoarce elementele într-un **ICollection**.

CollectionBase implementează toate metodele definite în interfețele **ICollection** și **IEnumerable**. Definește de asemenea o serie de metode protected ce pot fi extinse în subclase.

ReadOnlyCollectionBase definește o proprietate publică **Count** ce întoarce numărul de elemente ce se găsesc în colecție; definește de asemenea proprietatea protected **InnerList** ce întoarce un **ArrayList** cu elementele din colecție.

ReadOnlyCollectionBase implementează toate metodele ce sunt definite în interfețele **ICollection** și **IEnumerable**. Spre deosebire de clasa **CollectionBase**, **ReadOnlyCollectionBase** nu definește și un set de metode protected. Acestea nu își au rolul, întrucât colecția este read-only, deci nu sunt definite operații de adăugare, ștergere etc.


Implementarea unei clase dicționar proprii folosind **DictionaryBase**

Clasa **DictionaryBase** implementează interfețele **IDictionary**, **ICollection** și **IEnumerable**, oferind funcționalitate de bază pe care o puteți extinde pentru a implementa o clasă dicționar

proprie, puternic tipată.


DictionaryBase definește o proprietate publică **Count** ce întoarce numărul de intrări ce sunt în dicționar. **DictionaryBase** de asemenea definește proprietatea **InnerHashtable** ce întoarce într-un **Hashtable** toate obiectele din dicționar; proprietatea protected **Dictionary** oferă acces asupra elementelor dintr-un **IDictionary**.

DictionaryBase implementează toate metodele ce sunt definite în **IDictionary**, **ICollection** și **IEnumerable**, definind în plus același set de metode protected pe care le oferă și **CollectionBase**. **DictionaryBase** definește de asemenea metoda **OnGet** ce întoarce elementul cu o anumită cheie și valoare.



Colecții generice

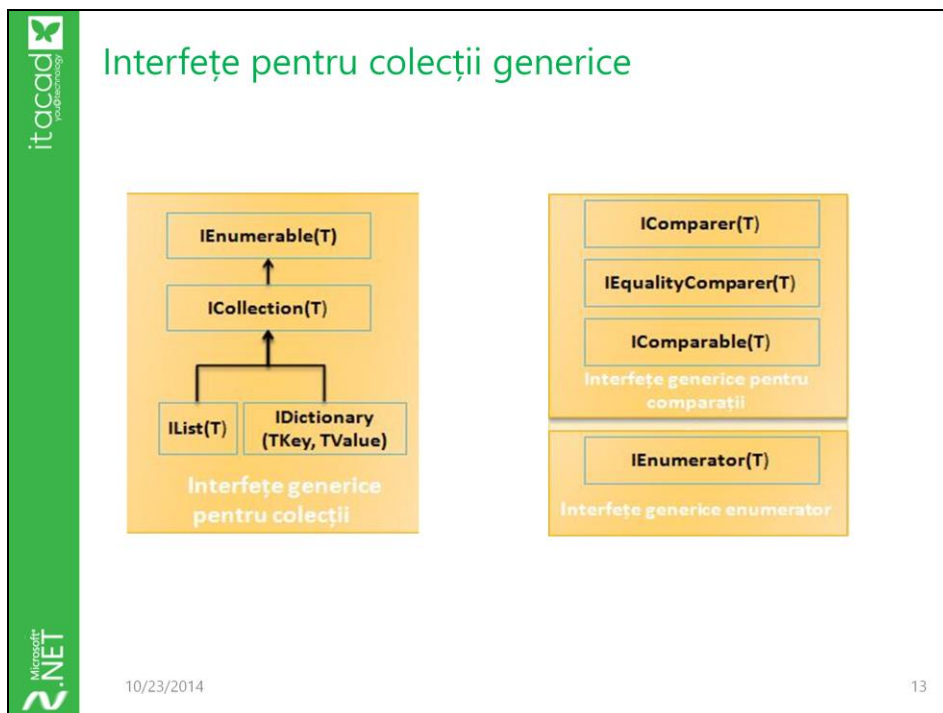
- ▶ Interfețe pentru colecții generice
- ▶ Folosirea colecției generice **List**
- ▶ Folosirea colecției generice **LinkedList**
- ▶ Folosirea colecțiilor generice **Queue** și **Stack**
- ▶ Folosirea colecțiilor generice tip **Dicționar**



10/23/2014 12



Colecțiile generice sunt puternic tipate. La crearea unei instanțe a unei colecții generice, trebuie specificat tipul elementelor ce vor fi reținute în colecție. De aceea, la adăugarea, ștergerea și accesarea elementelor din colecție, veți folosi tipuri specifice de date. Funcționarea acestor colecții este în contrast cu aceea a colecțiilor nongenerice, orientate pe obiecte, ce rețin referințe la **System.Object**.

Clasele colecție generice sunt grupate în namespace-ul **System.Collections.Generic**. Majoritatea claselor generice oferă funcționalitate similară echivalenților non-generici ce se găsesc în **System.Collections**. Pentru aplicații ce folosesc versiunea 2.0 sau mai nouă a platformei .NET, se recomandă utilizarea claselor colecție generice, din motive de performanțe.



Namespace-ul **System.Collections.Generic** definește interfețe generice ce specifică comportamentul pentru majoritatea colecțiilor generice în platforma .NET. Aceste interfețe pot fi organizate în trei grupuri, după cum urmează:

- Interfețe generice pentru colecții
- Interfețe generice pentru comparații
- Interfețe generice enumerator



Interfețe generice pentru colecții

- **IEnumerable**
 - **GetEnumerator()**
- **ICollection**
- **IList**
- **IDictionary**

Interfețe generice pentru comparații

- **IComparer**
 - **Compare()**
- **IEqualityComparer**
 - **Equals()**
 - **GetHashCode()**

10/23/2014 14

Namespace-ul **System.Collections.Generic** definește interfețe generice ce specifică comportamentul pentru majoritatea colecțiilor generice în platforma .NET. Aceste interfețe pot fi organizate în trei grupuri, după cum urmează:

- Interfețe generice pentru colecții
- Interfețe generice pentru comparații
- Interfețe generice enumerator

Interfețe generice pentru colecții

• **IEnumerable**

Definește o metodă **GetEnumerator** ce întoarce un obiect ce implementează interfața generică **IEnumerator**. Puteți folosi enumeratorul pentru a itera prin elementele colecției

• **ICollection**

Moștenește de la interfața generică **IEnumerable** și de la interfața nongenerică **IEnumerator**, și definește proprietăți și metode ce pot fi folosite cu toate colecțiile generice.

• **IList**

Moștenită de la interfața generică **ICollection** și reprezintă o colecție de obiecte ce pot fi accesate individual printr-un index. Pe lângă proprietățile ce sunt definite în interfața **ICollection**, **IList** definește o proprietate **Item** ce întoarce sau setează un element folosind un index zero-based.

• **IDictionary**

Moștenită din interfața generică **ICollection** și reprezintă o colecție de perechi cheie/valoare.

Fiecare element dintr-un obiect **IDictionary** este un obiect **KeyValuePair**.

Perechea cheie/valoare este o structură generică ce reprezintă o cheie de tipul specificat și o valoare de tipul specificat. Pentru a accesa obiectele cheie și valoare, folosiți proprietățile **Key** și **Value**

Interfețe generice pentru comparații

• **IComparer**

Definește o metodă **Compare** ce compară două obiecte de tipul precizat și întoarce o valoare ce indică dacă unul este egal, mai mic sau mai mare decât celălalt. Implementarea va trebui să



specifice ce înseamnă că un obiect este egal, mai mic sau mai mare decât altul. Clasa generică **List** conține metodele **Sort** și **BinarySearch** ce primesc ca parametru o instanță a unei clase ce implementează **IComparer** pentru a defini operația de comparare pentru elementele colecției. Implementarea implicită a interfeței generice **IComparer** este clasa generică **Comparer**. Pentru operații pe stringuri, folosiți clasa **StringComparer** în loc de **Comparer<String>**.

StringComparer definește metode și proprietăți ce permit executarea de comparații pe stringuri folosind diferite combinații de informații localizate și case-sensitivity.

- **IEqualityComparer**

Definește o metodă **Equals** ce determină dacă două obiecte specificate sunt egale. Interfața definește de asemenea o metodă **GetHashCode** ce întoarce codul hash al obiectului specificat.

Clasa generică **Dictionary** are constructori ce acceptă ca parametru o instanță a unei clase ce implementează interfața **IEqualityComparer** pentru a specifica cum va compara dicționarul cheile pentru egalitate. Implementarea implicită a interfeței generice este clasa generică **EqualityComparer**. Pentru egalități între stringuri folosiți clasa **StringComparer** în loc de **EqualityComparer<String>**



Interfețe generice pentru comparații între obiecte

- **IComparable**
 - **CompareTo ()**

Interfețe generice enumerator

- **IEnumerator**
 - **Compare()**
 - **MoveNext()**
 - **Reset()**

10/23/2014 15

Namespace-ul **System** definește o interfață generică ce permite unui obiect să se compare cu un altul de același tip, implementând interfața:

- **IComparable**

Definește o metodă **CompareTo** ce compară obiectul curent cu altul și întoarce o valoare pentru a indica dacă obiectul curent este mai mic, egal sau mai mare decât obiectul oferit. Interfață generică **IComparable** este implementată de acele tipuri ale căror valoare poate fi ordonată, cum ar fi tipurile numerice sau stringurile. Puteți implementa interfața generică **IComparable** pe propriile clase sau structuri pentru a crea metode ce execută comparații pe noi tipuri, de obicei în momentul în care se dorește sortarea.



Interfețe generice enumerator

Namespace-ul **System.Collections.Generic** definește o singură interfață generică enumerator:

- **IEnumerator**

Suportă enumerația peste o colecție generică. Se obține o instanță a unei clase ce implementează interfața generică **IEnumerator** când se invocă metoda **GetEnumerator** pe o colecție generică **ICollection**.

Interfața generică **IEnumerator** definește o proprietate **Current** ce indică elementul curent din colecție. Interfața generică **IEnumerator** moștenește metodele **MoveNext** și **Reset** de la interfața nongenerică **IEnumerator**.



Folosirea colecției generice **List**

- **List** este o colecție puternic tipată, ce păstrează ordinea elementelor
 - Elementele sunt accesate printr-un index *zero-based*
 - Predicate, Action

Exemplu:

```
List<DateTime> aList = ... ;  
aList.Add(aDateTime);  
DateTime d = aList[0];  
  
aList.Sort();  
int idx = aList.BinarySearch(d);
```

10/23/2014 16

Clasa generică **List** este o listă puternic tipată de obiecte ce pot fi accesate prin index și oferă metode pentru căutarea, sortarea și manipularea listelor. **List** implementează interfața generică **ICollection** folosind un vector a cărui dimensiune este mărită dinamic după cum este nevoie.

Folosirea clasei generice **List**

Clasa **List** este echivalentul generic al clasei nongenerice **ArrayList**. La stocarea unor tipuri referință, comportamentul celor două clase este identic. La stocarea tipurilor valoare, performanțele clasei **List** sunt mai bune deoarece nu mai sunt necesare operațiile de boxing și unboxing la adăugarea, ștergerea sau accesarea elementelor.

Metode:

// Obținerea primelor 3 elemente



```
List<DateTime> firstThree = aList.GetRange(0, 3);
```

// Ștergerea elementelor 3 și 4

```
aList.RemoveRange(2, 2);
```

// Căutarea unui element

```
int index = aList.BinarySearch(new DateTime(1968, 4, 5));
```

Obiecte generice **Predicate** și **Action**

Câteva dintre metodele clasei generice **List** pot primi parametrii **Predicate** sau **Action** pentru a oferi comportamentul următor:

- Un obiect generic **Predicate** încapsulează o operație ce ia un element al vectorului ca parametru, execută un test și întoarce o valoare Booleană. Un obiect **Predicate** este folosit pentru metode ca **Find** sau **TrueForAll**.
- Un obiect generic **Action** încapsulează o operație ce primește un element al vectorului ca parametru și execută o operație pe un obiect.

Exemplu:

```
// Găsirea tuturor datelor din 1960
List<DateTime> sixtiesDates = aList.FindAll(IsSixties);
// Metoda ce este încapsulată de Predicate în apelul de metodă FindAll
public bool IsSixties(DateTime dt)
{
    if (dt.Year >= 1960 && dt.Year < 1970)
        return true;
    else
        return false;
}

// Executarea unei operații pentru fiecare element
sixtiesDates.ForEach(IncrementYear);
// Metoda ce este încapsulată de Action în apelul de metoda ForEach
public void IncrementYear(DateTime dt)
{
    dt.AddYears(1);
}
```

10/23/2014
17

Câteva dintre metodele clasei generice **List** pot primi parametrii **Predicate** sau **Action** pentru a oferi comportamentul următor:

- Un obiect generic **Predicate** încapsulează o operație ce ia un element al vectorului ca parametru, execută un test și întoarce o valoare Booleană. Un obiect **Predicate** este folosit pentru metode ca **Find** sau **TrueForAll**.
- Un obiect generic **Action** încapsulează o operație ce primește un element al vectorului ca parametru și execută o operație pe un obiect. Folosiți un obiect **Action** pentru metoda **ForEach**

Următorul exemplu de cod exemplifică folosirea celor două:

```
// Găsirea tuturor datelor din 1960
List<DateTime> sixtiesDates = aList.FindAll(IsSixties);

// Executarea unei operații pentru fiecare element
sixtiesDates.ForEach(IncrementYear);
...
// Method that is encapsulated by a Predicate in the FindAll method call.
// Metoda ce este încapsulată de Predicate în apelul de metodă FindAll
public bool IsSixties(DateTime dt)
{
    if (dt.Year >= 1960 && dt.Year < 1970)
        return true;
}
```

```
else
```

```
    return false;
```

```
}
```



/ Metoda ce este încapsulată de Action în apelul de metoda ForEach

```
public void IncrementYear(DateTime dt)
```

```
{
```

```
    dt.AddYears(1);
```

```
}
```

Folosirea colecției generice **LinkedList**

- **LinkedList** este o listă dublu înlănțuită, puternic tipată
 - Nodurile se accesează secvențial

Exemplu:

```
LinkedList<DateTime> aLinkedList = ... ;
aLinkedList.AddFirst(aDateTime);

LinkedListNode<DateTime> node = aLinkedList.Find(aDateTime);
```

10/23/2014
18

Clasa generică **LinkedList** reprezintă o listă de obiecte puternic tipată, dublu înlănțuită, ce poate fi parcursă secvențial în orice direcție. Spre deosebire de clasa generică **List**, clasa **LinkedList** nu oferă metode pentru căutarea și sortarea metodelor, sau accesarea elementelor printr-un index.

De obicei, această clasă se folosește când doriți să rețineți obiecte puternic tipate în ordine, fără a avea nevoie să le accesați folosind un index

Fiecare nod dintr-o astfel de listă este de tipul **LinkedListNode**. **LinkedList** are definite proprietățile de **First** și **Last** pentru a indica primul și ultimul nod.

Fiecare obiect **LinkedListNode** dintr-o listă are definite proprietățile **Next** și **Previous** ce indică nodurile următor și precedent. Dacă aceste noduri nu există, cele două proprietăți vor întoarce valoarea null. Fiecare nod are o proprietate puternic tipată numită **Value** ce întoarce valoarea stocată în nod.

// Adăugarea unui nou element la start și reținerea referinței către nod

```
LinkedListNode<DateTime> node1 = aLinkedList.AddFirst(new
DateTime(1968, 4, 5));
```

// Adăugarea unui nod înainte de node1

```
aLinkedList.AddBefore(node1, new DateTime(1997, 7, 2));
```

// Eliminarea de elemente

```
aLinkedList.Remove(new DateTime(1965, 1, 19));
```

```
aLinkedList.RemoveLast();
```

// Iterarea prin toate elementele din listă

```
foreach (DateTime dt in aLinkedList)
```

```
{
```

```
        Console.WriteLine("{0}",  
dt.ToShortDateString());  
}
```

```
// Obținerea nodului următor
```



```
    LinkedListNode<DateTime> nextNode = node1.Next;
```

```
// Obținerea nodului anterior
```

```
    LinkedListNode<DateTime> previousNode =  
node1.Previous;
```

```
// Găsirea unui element
```

```
LinkedListNode<DateTime> foundNode =  
aLinkedList.Find(new DateTime(1968, 4, 5));
```



Folosirea colecțiilor generice **Queue** și **Stack**

➤ Queue este o colecție puterinic tipată, FIFO

Exemplu:

```
Queue<DateTime> aQueue = new Queue<DateTime>();

// Metodele Enqueue și Dequeue vor face verificări asupra tipului
aQueue.Enqueue(new DateTime(2008, 1, 19)); // Item 1.
DateTime item1 = aQueue.Dequeue();

// Obținerea lui Item 1 fără a îl extrage
DateTime item = aQueue.Peek();

// Reducerea capacității cozii
aQueue.TrimExcess();

// Iterarea prin elementele cozii folosind un Queue.Enumerator.
Queue<DateTime>.Enumerator qenum = aQueue.GetEnumerator();
while (qenum.MoveNext())
{
    DateTime dt = qenum.Current;
    Console.WriteLine("{0}", dt.ToShortDateString());
}
```

10/23/2014 19

Clasa generică **Queue** reprezintă o colecție FIFO de obiecte de un anume tip. Clasa generică **Stack** reprezintă o colecție LIFO de obiecte de un anume tip.



Dacă creați o aplicație ce folosește platformat .NET 2.0 sau mai recentă, se recomandă folosirea claselor **Stack** și **Queue** generice în locul echivalentului lor nongeneric.

Ambele clase implementează interfața generică **IEnumerable** și interfața nongenerică **ICollection**.

Folosirea clasei generice **Queue**

Această clasă implementează o coadă ca un vector circular în care elementele se adaugă pe la un capăt și extrase din celălalt.

Clasa generică **Queue** implementează toate metodele și proprietățile interfețelor sale de bază



Folosirea colecțiilor generice **Queue** și **Stack**

➤ Stack este o colecție puternic tipată, LIFO

Exemplu:

```
// Instanțierea unei stive de obiecte DateTime
Stack<DateTime> aStack = new Stack<DateTime>();
aStack.Push(new DateTime(2008, 1, 19)); // Item 1.
aStack.Push(new DateTime(2008, 4, 5)); // Item 2.

// Obținerea Item 2 fără a îl extrage
DateTime item = aStack.Peek();

// Extragerea Item 2
DateTime item2 = aStack.Pop();

// Iterarea prin stivă folosind un Stack.Enumerator.
Stack<DateTime>.Enumerator stkenum = aStack.GetEnumerator();
while (stkenum.MoveNext())
{
    DateTime dt = stkenum.Current;
    Console.WriteLine("{0}", dt.ToShortDateString());
}
```



10/23/2014 20

Folosirea clasei generice **Stack**

Această clasă implementează o stivă ca un vector circular, în care elementele sunt inserate și extrase prin același capăt.

Clasa generică **Stack** implementează toate metodele și proprietățile interfețelor de bază.

```
// Iterarea prin Stack folosind instrucțiunea foreach
foreach (DateTime dt in aStack)
{
    Console.WriteLine("{0}", dt.ToShortDateString());
}
```



Folosirea colecțiilor generice tip **Dictionar**

- **Dictionary**: colecție tip dicționar puternic tipată
 - Echivalent cu Hashtable
- **SortedDictionary**: reține informația sub forma unui arbore binar
- **SortedList**: vector puternic tipat de perechi cheie/valoare

Exemplu:

```
Dictionary<string, double> dict = ... ;
dict.Add("Chai", 18.0);
double val = dict["Chai"];

// Eliminarea intrării din dicționar
dict.Remove("Chai");

// Iterarea folosind o instrucțiune foreach
foreach (KeyValuePair<string, double> entry in dict)
{
    string k = entry.Key;
    double v = entry.Value;
}
```

10/23/2014 21

Fiecare clasă generică tip dicționar primește două tipuri de parametru, TKey și TValue, ce specifică tipurile de date pentru cheile și valorile din dicționar.

Folosirea clasei generice **Dictionary**

Această clasă implementează toate metodele și proprietățile ce sunt definite în interfețele de bază. Pe lângă acestea, definește o metodă **ContainsValue** ce indică dacă dicționarul conține o anumită valoare.



Folosirea clasei generice **SortedDictionary**

Această clasă generică definește metode și proprietăți similare clasei generice **Dictionary** dar va stoca elementele într-un arbore binar în loc de un hashtable.

La accesarea cheilor sau valorilor dintr-un **SortedDictionary** folosind proprietățile **Keys** sau **Values**, sau când iterați prin **SortedDictionary** folosind un enumerator, este garantat că veți găsi elementele în ordine, bazat pe comparațiile dintre chei.

Folosirea clasei generice **SortedList**

Această clasă generică definește metode și proprietăți similare claselor de mai sus. Următorul tabel prezintă metodele în plus pe care aceasta le oferă.



Colecții specializate

- Folosirea colecțiilor specializate pe **String-uri**
- Folosirea colecțiilor specializate de tip dicționar
- Folosirea colecțiilor cu nume
- Folosirea clasei **CollectionsUtil**
- Folosirea structurii **BitVector32**

10/23/2014 22

Namespace-ul **System.Collections.Specialized** conține colecții specializate și puternic tipate, cum ar fi dicționare tip listă înlănțuită, un vector de biți sau colecții ce conțin numai string-uri. Această lecție descrie colecțiile din acest namespace și explică folosirea lor

Obiective:

La finalul acestei lecții, veți putea să:

- ☐ Folosiți colecții specializate de string-uri
- ☐ Folosiți colecții dicționar specializate
- ☐ Folosiți colecții cu nume
- ☐ Creați instanțe ale claselor **Hashtable** și **SortedList** ce nu sunt case-sensitive
- ☐ Folosiți colecții tip vector de biți

Folosirea colecțiilor specializate pe String-uri

- **StringCollection** este o listă de string-uri înlănțuite
- **StringDictionary** este un dicționar în care cheile și valorile sunt de tip string

Exemplu **StringCollection**:

```
StringCollection strCol = new StringCollection();  
// Adăugarea unui interval dintr-un vector la finalul  
colecției  
string[] myArr = {"Smith", "Jones", "Martinez", "Smith"};  
strCol.AddRange(myArr);  
// Adăutarea, inserția și extragerea string-urilor  
strCol.Add("Martinez");  
strCol.Insert(3, "Rangel");  
strCol.Remove("Jones");  
// Afișarea primului string din colecție  
string str = strCol[0];  
Console.WriteLine("First string is {0}.", str);
```

10/23/2014

23

Namespace-ul **System.Collections.Specialized** definește următoarele colecții specializate de stringuri:

- **StringCollection**
- **StringDictionary**

Folosirea clasei **StringCollection**

Această clasă reprezintă o listă secvențială de stringuri. Puteți adăuga stringuri în orice poziție în colecție și puteți accesa elementele de tip string prin indicele lor zero-based.



Folosirea colecțiilor specializate pe String-uri

Exemplu **StringDictionary** :

```
// Crează un StringDictionary gol și adaugă perechi  
chei/valoare  
StringDictionary strDict = new StringDictionary();  
strDict.Add("NY", "New York");  
// Stabilire dacă un StringDictionary conține o cheie  
specifică  
if (strDict.ContainsKey("TX"))  
{  
    string val = strDict["TX"];  
    Console.WriteLine("Key 'TX' has value {0}.", val);  
}  
// Stabilire dacă StringDictionary conține o anumită valoare  
if (strDict.ContainsValue("California"))  
{  
    Console.WriteLine("The dictionary contains the value  
'California'.");  
}
```

Folosirea clasei **StringDictionary**

Această clasă reprezintă un hashtable în care atât cheile cât și valorile sunt de tipul string.



Folosirea colecțiilor specializate de tip dicționar

- **ListDictionary** – implementează un dicționar ca o listă simplu înlănțuită
- **HybridDictionary** – implementează un dicționar ca o listă sau ca un hash table, depinzând de dimensiune
- **OrderedDictionary** – implementează un dicționar ale cărui elemente sunt accesibile prin cheie sau index

10/23/2014 25

Folosirea clasei ListDictionary

Această clasă implementează interfața **IDictionary** ca o listă simplu înlănțuită. Folosiți **ListDictionary** pentru colecții ce conțin 10 sau mai puține intrări deoarece este mai rapidă și mai mică decât un **Hashtable** în acest caz. Dacă trebuie să stocați mai mult de 10 intrări, folosiți un obiect de tip **Hashtable**.

ListDictionary este implementată pentru accesări rapide pe bază de cheie, ordinea internă a elementelor nu este specificată. Performanțele metodelor și proprietăților obiectelor **ListDictionary** sunt proporționale cu numărul de elemente din colecție.

ListDictionary are proprietăți standard ale dicționarelor cum ar fi **Item**, **Keys** și **Values**, precum și metode standard precum **Add**, **Clear**, **GetEnumerator** și **Remove**. Modul de folosire este același cu cel pentru **Hashtable**.

Folosirea clasei HybridDictionary

Această clasă implementează interfața **IDictionary** folosind un obiect **ListDictionary** când numărul de elemente din colecție este mic și trecând automat la **Hashtable** când acesta crește.

Folosiți această clasă în cazul în care numărul de elemente al dicționarului nu este cunoscut. **HybridDictionary** profită atât de performanțele sporite ale **ListDictionary** în momentul în care numărul de elemente este mic, precum și de capacitatea **Hashtable** de a lucra cu un număr mare de elemente în momentul în care este cazul. Dacă numărul de elemente oferit la creare este suficient de mare, se va folosi de la început un **Hashtable** pentru a stoca informația, nemaifiind nevoie să se facă o copiere din obiectul **ListDictionary**.

HybridDictionary are proprietăți standard ale dicționarelor cum ar fi **Item**, **Keys** și **Values**, precum și metode standard precum **Add**, **Clear**, **GetEnumerator** și **Remove**. Modul de folosire este același cu cel pentru **Hashtable**.



Folosirea colecțiilor specializate de tip dicționar

Exemplu **OrderedDictionary**:

```
// Crearea unui OrderedDictionary gol și adăugarea de intrări cheie/valoare
OrderedDictionary ordDict = new OrderedDictionary();
ordDict.Add("chai", 18.0);

// Obținerea unei intrări prin cheie și index
Console.WriteLine("Value of 'chai': {0:c}", ordDict["chai"]);
Console.WriteLine("Value of item 0: {0:c}", ordDict[0]);

// Inserția și ștergerea unei intrări prin index
ordDict.Insert(2, "Northwoods Cranberry Sauce", 19.95);
ordDict.RemoveAt(1);

// Determinarea dacă OrderedDictionary conține o cheie specifică
if (ordDict.Contains("chai"))
{
    double val = (double)ordDict["chai"];
    Console.WriteLine("Chai price: {0:c}.", val);
    // Ștergerea intrării prin cheie
    ordDict.Remove("chai");
}
```



10/23/2014

26

Folosirea clasei **OrderedDictionary**

Această clasă implementează interfețele **IDictionary** și **IOrderedDictionary**. **OrderedDictionary** reprezintă o colecție de perechi cheie/valoare ce sunt accesibile fie prin cheie, fie printr-un index. Elementele unui astfel de obiect nu sunt sortate în nici un fel.

OrderedDictionary are proprietăți standard ale dicționarelor cum ar fi **Item**, **Keys** și **Values**. Proprietatea **Item** este supraîncărcată pentru a putea permite accesarea unui element fie prin cheie, fie prin index.



Folosirea colecțiilor cu nume

- **NamedObjectCollectionBase**
 - Clasă de bază abstractă
 - Reprezintă o colecție de chei stringuri și valori obiecte
 - Elementele sunt accesibile fie prin cheie, fie prin index
- **NamedValueCollection**
 - Moștenită din **NamedObjectCollectionBase**
 - Reprezintă o colecție de chei string-uri și valori string-uri
 - Permite stocarea mai multor valori de tip string pentru o cheie

10/23/2014 27

Folosirea clasei **NamedObjectCollectionBase**

Aceasta clasă abstractă reprezintă o colecție de chei string-uri și valori obiecte ce pot fi accesate fie prin cheie, fie prin index. **NamedObjectCollectionBase** reține datele într-un hash table, fiecare intrare în hash table fiind o pereche cheie/valoare

Folosirea clasei **NameValueCollection**

Această clasă moștenește **NameObjectCollectionBase** și reprezintă o colecție de chei stringuri și valori stringuri. **NameValueCollection** permite reținerea a mai multor valori stringuri pentru aceeași cheie și este potrivită dacă doriți să rețineți colecții cum ar fi date obținute dintr-un formular sau string-uri pentru diverse interogări.



Folosirea colecțiilor cu nume

Exemplu **NameValueCollection**:

```
// Crearea unui NameValueCollection gol și adăugarea de valori multiple pentru fiecare cheie
NameValueCollection nvCol = new NameValueCollection();
nvCol.Add("Minnesota", "St. Paul");
nvCol.Add("Minnesota", "Minneapolis");

// Obținerea cheii cu indexul 0 (Minnesota)
string key = nvCol.GetKey(0);

// Ștergerea intrării pentru Minnesota prin cheie
nvCol.Remove(key);

// Obținerea valorilor pentru cheia Florida ca un vector de string-uri
foreach (string val in nvCol.GetValues("Florida"))
{
    Console.WriteLine("A value for 'Florida': {0}", val);
}

// Iterarea prin NameValueCollection
foreach (string k in nvCol)
{
    // Obținerea valorilor pentru o intrare ca o listă separată prin virgulă
    Console.WriteLine("Key: {0} Values: {1}", k, nvCol.Get(k));
}
```

10/23/2014

28

Folosirea clasei **NameValueCollection**

Această clasă moștenește **NameObjectCollectionBase** și reprezintă o colecție de chei stringuri și valori stringuri. **NameValueCollection** permite reținerea a mai multor valori stringuri pentru aceeași cheie și este potrivită dacă doriți să rețineți colecții cum ar fi date obținute dintr-un formular sau string-uri pentru diverse interogări.

Folosirea clasei **CollectionsUtil**

➤ **CollectionsUtil** permite crearea unui obiect *Hashtable* sau *SortedList* ce nu este case-sensitive

Exemplu **CollectionsUtil** :

```
// Crează un Hashtable non-case-sensitive, adaugă un element și îl accesează
Hashtable ht = CollectionsUtil.CreateCaseInsensitiveHashtable();
ht.Add("UK", "London");
Console.WriteLine("Capital of UK: {0}", ht["uk"]);

// Crează un SortedList non-case-sensitive, adaugă un element și îl accesează
SortedList sl = CollectionsUtil.CreateCaseInsensitiveSortedList();
sl.Add("USA", "Washington D.C.");
Console.WriteLine("Capital of USA: {0}", sl["usa"]);
```

Clasa **CollectionUtil** din namespace-ul **System.Collections.Specialized** crează colecții ce nu fac diferența între literele mari și literele mici într-un string. Clasa conține metode statice ce generează instanțe non-case-sensitive ale colecției ce folosește implementări non-case-sensitive ale unui hashtable și comparator.

Folosirea structurii **BitVector32**

- **BitVector32** stochează valori Booleane sau întregi de dimensiuni mici
- Pentru a stoca valori Booleane
 - Se apelează metoda **CreateMask** pentru a crea o mască
 - Se accesează un bit folosind masca
- Pentru a stoca valori întregi de dimensiuni mici:
 - Se apelează metoda **CreateSection** pentru a crea o **BitVector32.Section**
 - Valorile se accesează folosind parametrul **BitVector32.Section**

Structura **BitVector32** din namespace-ul **System.Collections.Specialized** stochează valori Booleane și întregi de dimensiune mică în zone de memorie de 32 de biți.

BitVector32 este mai eficient decât **BitArray** pentru valori Booleane ce sunt folosite intern într-o aplicație. Un obiect **BitArray** poate crește nedefinit și are penalități de preformanță datorate faptului că este instanța unei clase, în timp ce **BitVector32** este garantat să rețină doar 32 biți.

Folosirea structurii **BitVector32**

Structura **BitVector32** poate fi setată să rețină întregi de dimensiune mică sau valori Booleane, dar nu pe amândouă odată.

Valoarea **BitVector32** este o fereastră în structura **BitVector32** și reprezintă numărul minim de biți succesivi ce conțin valoarea maximă ce este specificată de către metoda **CreateSection**. De exemplu, o secțiune ce are valoarea maximă 5 este compusă din 3 biți.

Folosirea structurii **BitVector32**

Exemplu **BitVector32** :

```
// Crearea unui BitVector32 cu toți biții setați pe false
BitVector32 bv = new BitVector32(0);

// Crearea unei măști pentru a izola primii 3 biți
int myBit1 = BitVector32.CreateMask();
int myBit2 = BitVector32.CreateMask(myBit1);
int myBit3 = BitVector32.CreateMask(myBit2);

// Setarea biților impari pe true
bv[myBit1] = true;
bv[myBit3] = true;

// Creare secțiuni cu valori maxime
BitVector32.Section mySect1 = BitVector32.CreateSection(6);
BitVector32.Section mySect2 = BitVector32.CreateSection(3, mySect1);

// Setare acțiune la o valoare
bv[mySect1] = 5;
bv[mySect2] = 3;
```

Best Practices

- Folosiți pe cât se poate clase existente pentru colecții
- Folosiți colecții generice doar când este nevoie
- Selectați colecția pe care o veți folosi în funcție de cerințele de performanță și viteză

Review

- Colecții orientate obiect
- Colecții generice
- Colecții specializate

