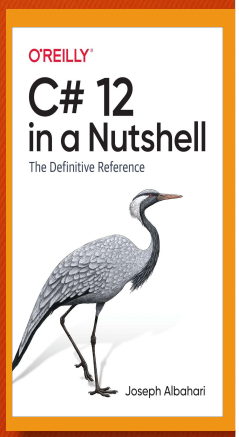
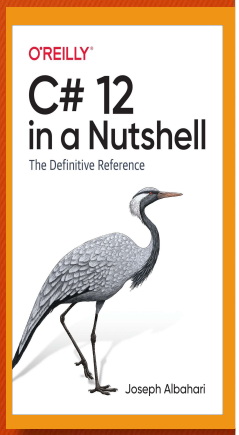
Chapter 7
Lists, Queues, Stacks, and Sets



- List<T> and ArrayList
- LinkedList<T>
- Queue<T> and Queue
- Stack<T> and Stack
- BitArray
- HashSet<T> and SortedSet<T>

List<T> and ArrayList



کلاس جنریک List و غیر جنریک ArrayList به ما اجازه میدن تا آرایه هایی از هر نوع با اندازه‌های غیر ثابت بسازیم و همچنین یکی از پرکاربردترین نوع‌های Collection ها هستند.

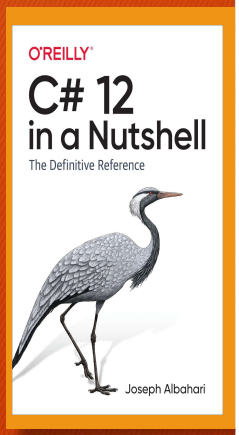
ArrayList اینترفیس IList رو پیاده‌سازی میکنه در صورتی که List<T> هر دو اینترفیس IList و IList<T> رو پیاده‌سازی میکنه.

برخلاف کلاس Array این دو کلاس همه متدهای اینترفیس‌ها رو به صورت public پیاده‌سازی کردند و متدهایی مثل Add یا Remove به همون شکلی که انتظار داریم کار میکنند.

این دو کلاس در اصل با یک آرایه داخلی کار میکنند که به محض پر شدن ظرفیتش با یک آرایه بزرگتر جایگزین میشه، در نتیجه امکان افزودن عنصر جدید همیشه وجود دارد چون همیشه حداقل خونه آخر این آرایه خالیست، ولی با این وجود افزودن و حذف عنصر میتونه سرعت پایینی داشته باشه چون تمامی عناصر لیست باید شیفت پیدا کنند.

List<T> اگر T یک نوع value-type باشه به مراتب سریعتر از ArrayList عمل میکنه چرا که نیاز به تبدیل نوع نداره.

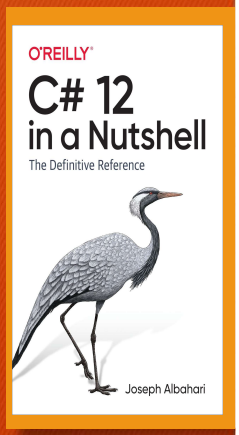
List<T> and ArrayList



هر دو کلاس ArrayList و List<T> سازنده‌هایی دارند که یک collection به عنوان ورودی میگیره و یه آبجکت جدید از اون میسازه. این کار همه عناصر collection موجود را در آبجکت جدید ذخیره میکنه.

```
List<int> numberList = new List<int>();
ArrayList numberArrays = new ArrayList(numberList);

numberList.Add(0);
Console.WriteLine(numberList.Count);    // 1
Console.WriteLine(numberArrays.Count);  // 0

List<DateTime> dateList = new List<DateTime>();
dateList.Add(DateTime.Now);
ArrayList dateArrays = new ArrayList(dateList);
dateArrays.Add(DateTime.Now.AddDays(1));

Console.WriteLine(dateList.Count);      // 1
Console.WriteLine(dateArrays.Count);    // 2
dateArrays[0] = DateTime.Now.AddDays(-1);

Console.WriteLine(dateList[0]);    //Today
Console.WriteLine(dateArrays[0]);  //Yesterday
```

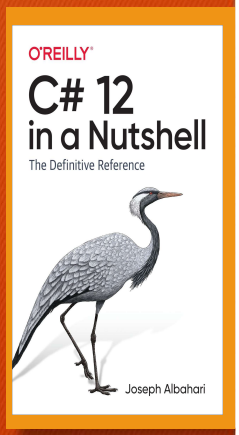
List<T> and ArrayList



مثال زیر کاربرد بخشی از متدهای موجود در این کلاس‌هاست:

```
var words = new List<string>();    // New string-typed list

words.Add("melon");
words.Add("avocado");
words.AddRange(["banana", "plum"]);
words.Insert(0, "lemon");                    // Insert at start
words.InsertRange(0, ["peach", "nashi"]);    // Insert at start

words.Remove("melon");
words.RemoveAt(3);                          // Remove the 4th element
words.RemoveRange(0, 2);                    // Remove first 2 elements

// Remove all strings starting in 'n':
words.RemoveAll(s => s.StartsWith("n"));

Console.WriteLine(words[0]);                          // first word
Console.WriteLine(words[words.Count - 1]);            // last word
foreach (string s in words) Console.WriteLine(s);      // all words
List<string> subset = words.GetRange(1, 2);            // 2nd->3rd words

string[] wordsArray = words.ToArray();    // Creates a new typed array

// Copy first two elements to the end of an existing array:
string[] existing = new string[1000];
words.CopyTo(0, existing, 998, 2);

List<string> upperCaseWords = words.ConvertAll(s => s.ToUpper());
List<int> lengths = words.ConvertAll(s => s.Length);
```

List<T> and ArrayList



کلاس غیرجنریک ArrayList نیاز به تبدیل‌های زیادی داره، این مثال رو ببینید:

```
ArrayList al = new ArrayList();
al.Add("hello");
string first = (string)al[0];
string[] strArr = (string[])al.ToArray(typeof(string));

int firstNumber = (int)al[0];     // Runtime exception
```

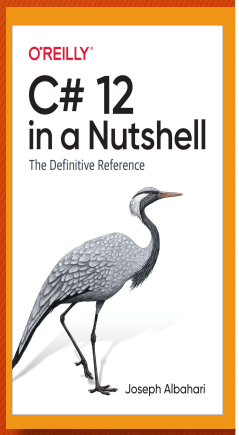
همچنین احتمال وقوع خطا هنگام اجرا نیز بالا میره.

ArrayList از نظر عملکرد شبیه List<object> هست و برای مواقعی که شما نیاز به لیستی از عناصر با نوع های مختلف دارید میتونه مفید باشه.

با استفاده از متدهای cast و ToList در فضای نام system.Linq میتونیم یک ArrayList رو به List<T> تبدیل کنیم.

```
ArrayList sampleArrayList = new ArrayList();
sampleArrayList.AddRange(new[] { 1, 5, 9 });
List<int> list = sampleArrayList.Cast<int>().ToList();
```

LinkedList<T>



LinkedList<T> is a generic doubly linked list.

(نمیدونستم چی ترجمه کنم اصل متن رو آوردم.)

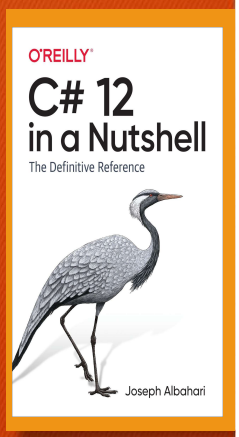
یک doubly linked list زنجیره‌ای از Node هاست هر کدوم به Node های قبل، بعد و عنصر اصلی رفرنس دارند.

مزیت اصلی این کالکشن اینه که شما یک عنصر رو هرجایی که بخواهید میتونید اضافه کنید زیرا فقط نیاز داره که یک node جدید ایجاد و چند رفرنس رو آپدیت کنه.

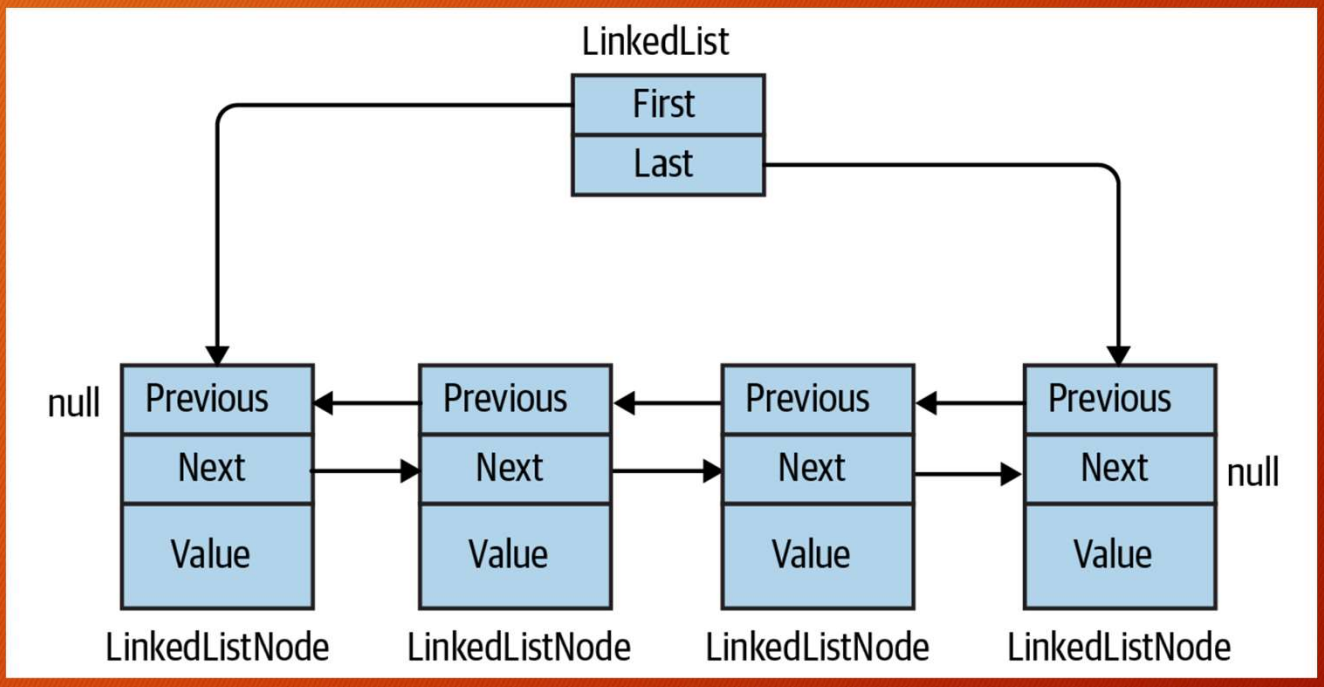
However, finding where to insert the node in the first place can be slow because there's no intrinsic mechanism to index directly into a linked list; each node must be traversed, and binary-chop searches are not possible.

(ممنون میشم ترجمه این بخش رو شما کمک کنید.)

LinkedList<T>

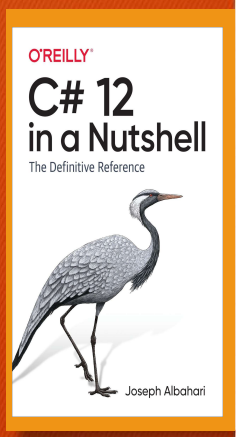


این تصویر رو ببینید:



LinkedList<T> اینترفیس‌های ICollection و Ienumerable و جنریک‌هاشون رو پیاده‌سازی کردند اما IList<T> رو پیاده‌سازی نمیکنه چون امکان دسترسی به عناصر اون توسط ایندکس وجود نداره.

LinkedList<T>

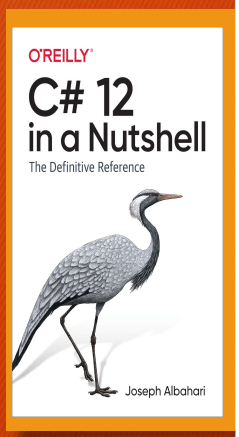


لیست node ها از طریق کلاس LinkedListNode<T> پیاده‌سازی میشه:

```
public sealed class LinkedListNode<T>
{
    internal LinkedList<T>? list;
    internal LinkedListNode<T>? next;
    internal LinkedListNode<T>? prev;
    internal T item;

    public LinkedListNode(T value)...
    internal LinkedListNode(LinkedList<T> list, T value)...

    public LinkedList<T>? List...
    public LinkedListNode<T>? Next...
    public LinkedListNode<T>? Previous...
    public T Value...
    ...public ref T ValueRef => ref item;

    internal void Invalidate()...
}
```

هنگام افزودن یک node میتونیم موقعیت اون رو نسبت به یک node دیگه یا ابتدا و انتهای لیست مشخص کنیم.

LinkedList<T>



در LinkedList<T> متدهای زیر برای افزودن عنصر جدید وجود دارد:

```
void ICollection<T>.Add(T value)...
public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value)...
public void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)...
public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value)...
public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)...
public LinkedListNode<T> AddFirst(T value)...
public void AddFirst(LinkedListNode<T> node)...
public LinkedListNode<T> AddLast(T value)...
public void AddLast(LinkedListNode<T> node)...
```

متد Add در داخل خود متد AddLast را فراخوانی میکنه.

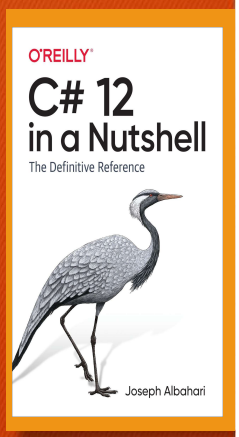
همچنین متدهای مشابهی برای حذف عنصر از لیست وجود داره:

```
public void Clear()...
public bool Remove(T value)...
public void Remove(LinkedListNode<T> node)...
public void RemoveFirst()...
public void RemoveLast()...
```

فیلدهای زیر هم میتونه خیلی بهمون کمک کنه:

```
public int Count...
public LinkedListNode<T>? First...
public LinkedListNode<T>? Last...
```

LinkedList<T>

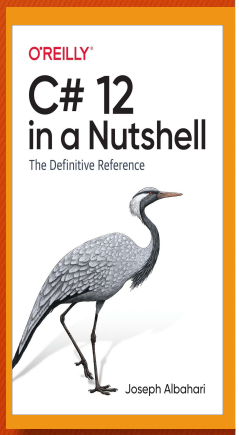


و در انتها متدهایی برای جستجو در لیست و کپی لیست در یک آرایه برای کار با index ها وجود دارد:

```
public bool Contains(T value)...
public void CopyTo(T[] array, int index)...
public LinkedListNode<T>? Find(T value)...
public LinkedListNode<T>? FindLast(T value)...
public Enumerator GetEnumerator() => new Enumerator(this);
```

متد GetEnumerator() برای استفاده از LinkrdList<T> در دستور foreach کاربرد دارد.

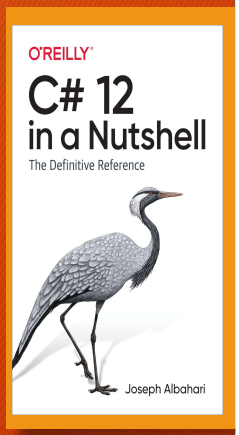
LinkedList<T>



یک مثال هم ببینیم و بریم سراغ بحث بعدی:

```
LinkedList<int> intLinkList = new LinkedList<int>();

var tune = new LinkedList<string>();
tune.AddFirst("do");                                // do
tune.AddLast("so");                                 // do - so

tune.AddAfter(tune.First, "re");                    // do - re- so
tune.AddAfter(tune.First.Next, "mi");               // do - re - mi- so
tune.AddBefore(tune.Last, "fa");                    // do - re - mi - fa- so

tune.RemoveFirst();                                  // re - mi - fa - so
tune.RemoveLast();                                   // re - mi - fa

LinkedListNode<string> miNode = tune.Find("mi");
tune.Remove(miNode);                                // re - fa
tune.AddFirst(miNode);                              // mi- re - fa

foreach (string s in tune) Console.WriteLine(s);
```

Queue<T> and Queue



Queue<T> and Queue یک ساختار داده FIFO هست،
(ازینجا به بعد صف صداش میکنیم)
متد Enqueue یک عنصر به انتهای صف اضافه میکنه و
متد Dequeue عنصر اول صف رو برمیگردونه و حذفش میکنه.
متد Peek بدون حذف عنصر اول، اون رو برمیگردونه.

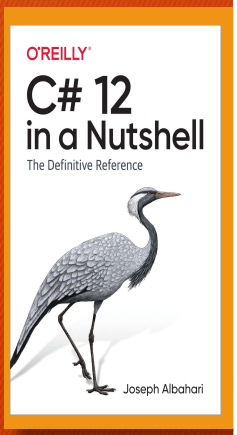
پراپرتی Count برای بررسی اینکه آیا عنصر وجود داره قبل از Dequeue میتونه مفید باشه.

همچنین صف‌ها enumerable هستند و اینترفیس IList رو پیاده‌سازی نمیکنند چون عناصرش از طریق ایندکس در دسترس نیستند.

البته با استفاده از متد ToArray میتونید یک صف رو به آرایه تبدیل کنید و به عناصرش به صورت تصادفی دسترسی داشته باشید.

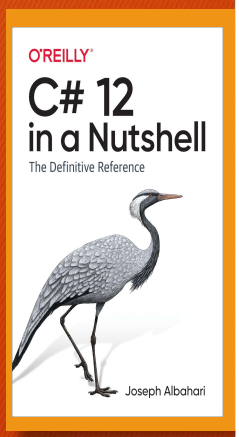
متدهای دیگری هم وجود دارد که در اسلاید بعد به طور کامل میتونید همشونو ببینید:

Queue<T> and Queue




```
public class Queue<T> : IEnumerable<T>,
    ICollection,
    IReadOnlyCollection<T>
{
    private T[] _array;
    private int _head;       // The index from which to dequeue if the queue isn't empty.
    private int _tail;       // The index at which to enqueue if the queue isn't full.
    private int _size;       // Number of elements.
    private int _version;

    public Queue()...
    public Queue(int capacity)...
    public Queue(IEnumerable<T> collection)...

    public int Count...
    bool ICollection.IsSynchronized...
    object ICollection.SyncRoot => this;

    public void Clear()...
    public void CopyTo(T[] array, int arrayIndex)...
    void ICollection.CopyTo(Array array, int index)...
    public void Enqueue(T item)...
    public Enumerator GetEnumerator() => new Enumerator(this);

    IEnumerator<T> IEnumerable<T>.GetEnumerator()...

    IEnumerator IEnumerable.GetEnumerator() => ((IEnumerable<T>)this).GetEnumerator();

    public T Dequeue()...
    public bool TryDequeue([MaybeNullWhen(false)] out T result)...
    public T Peek()...
    public bool TryPeek([MaybeNullWhen(false)] out T result)...
    public bool Contains(T item)...
    public T[] ToArray()...
    private void SetCapacity(int capacity)...
    private void MoveNext(ref int index)...
    private void ThrowForEmptyQueue()...
    public void TrimExcess()...
    public int EnsureCapacity(int capacity)...
    private void Grow(int capacity)...

    public struct Enumerator ...
}
```

Queue<T> and Queue

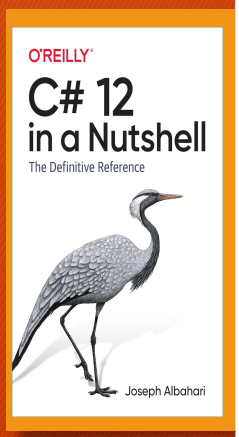


صف در داخل خود (مثل List<T>) یک آرایه دارند که در صورت نیاز میتونه سایزش تغییر کنه.

صف ایندکس‌های عناصر ابتدایی و انتهایی رو نگه میداره به همین علت enqueuing و dequeuing با سرعت بالایی اتفاق میفته. (مگر در مواردی که نیاز به تغییر سایز آرایه داخلی باشد.)

```
var q = new Queue<int>();
q.Enqueue(10);
q.Enqueue(20);
int[] data = q.ToArray();          // Exports to an array
Console.WriteLine(q.Count);       // "2"
Console.WriteLine(q.Peek());      // "10"
Console.WriteLine(q.Dequeue());   // "10"
Console.WriteLine(q.Dequeue());   // "20"
Console.WriteLine(q.Dequeue());   // throws an exception (queue empty)
```

Stack<T> and Stack



Stack<T> and Stack یک ساختار داده LIFO هست.

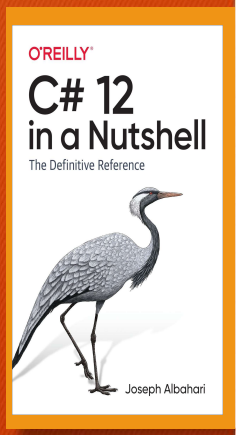
متد Push یک عنصر به بالای stack اضافه میکنه.
متد POP عنصر بالای stack را برمیگردونه و حذفش میکنه.
متد Peek عنصر بالای stack رو بدون حذف کردن، برمیگردونه.

پراپرتی Count تعداد عناصر stack رو برمیگردونه.

متد ToArray یک stack رو تبدیل به آرایه میکنه تا بتونیم به صورت تصادفی به ایندکس‌های مختلف اون دسترسی داشته باشیم.

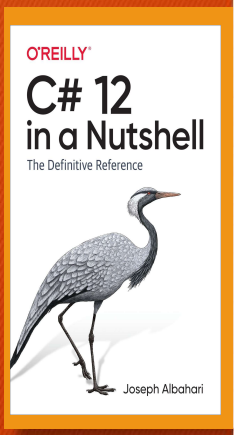
در اسلاید بعد میتونید چگونگی تعریف کلاس Stack رو ببینید:

Stack<T> and Stack




```
...public class Stack<T> : IEnumerable<T>,
    System.Collections.ICollection,
    IReadOnlyCollection<T>
{
    private T[] _array; // Storage for stack elements. Do not rename (binary serialization)
    private int _size; // Number of items in the stack. Do not rename (binary serialization)
    private int _version; // Used to keep enumerator in sync w/ collection. Do not rename (binary serialization)

    private const int DefaultCapacity = 4;

    public Stack()...
    ...public Stack(int capacity)...
    ...public Stack(IEnumerable<T> collection)...

    public int Count...
    bool ICollection.IsSynchronized...
    object ICollection.SyncRoot => this;

    ...public void Clear()...
    public bool Contains(T item)...
    ...public void CopyTo(T[] array, int arrayIndex)...
    void ICollection.CopyTo(Array array, int arrayIndex)...
    ...public Enumerator GetEnumerator() => new Enumerator(this);

    ...IEnumerator<T> IEnumerable<T>.GetEnumerator()...

    IEnumerator IEnumerable.GetEnumerator() => ((IEnumerable<T>)this).GetEnumerator();

    public void TrimExcess()...
    ...public T Peek()...
    public bool TryPeek([MaybeNullWhen(false)] out T result)...
    ...public T Pop()...
    public bool TryPop([MaybeNullWhen(false)] out T result)...
    ...public void Push(T item)...
    ...private void PushWithResize(T item)...
    ...public int EnsureCapacity(int capacity)...
    private void Grow(int capacity)...
    ...public T[] ToArray()...
    private void ThrowForEmptyStack()...

    public struct Enumerator ...
}
```

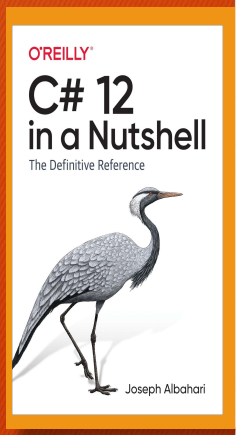
Stack<T> and Stack



Stack هم مثل Queue و List یک آرایه داخلی داره که در صورت نیاز اونو تغییر سایز میده.

```
var s = new Stack<int>();
s.Push(1);                      //           Stack = 1
s.Push(2);                      //           Stack = 1,2
s.Push(3);                      //           Stack = 1,2,3
Console.WriteLine(s.Count);     // Prints 3
Console.WriteLine(s.Peek());    // Prints 3,  Stack = 1,2,3
Console.WriteLine(s.Pop());     // Prints 3,  Stack = 1,2
Console.WriteLine(s.Pop());     // Prints 2,  Stack = 1
Console.WriteLine(s.Pop());     // Prints 1,  Stack = <empty>
Console.WriteLine(s.Pop());     // throws exception
```

BitArray



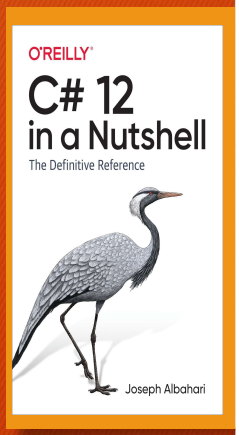
BitArray یک کالکشن با سایز متغیر هست که عناصر آن bool هستند. استفاده از این کالکشن از نظر اشغال حافظه به صرفه تر از یک آرایه معمولی با عناصر bool یا یک List<bool> است چرا که BitArray برای هر مقدار یک بیت از حافظه را اشغال میکند ولی عناصر لیست از جنس bool هر کدام یک بایت از حافظه را اشغال میکنند.

با استفاده از indexer میتونیم یک عنصر از bitArray را بخوانیم یا تغییر دهیم.

همچنین این کالکشن چهار عملگر برای کار با بیت ها در اختیارمون میذاره: or, and, xor, not

```
BitArray bitArray = new BitArray(new int[] { 10, 20 });
Console.WriteLine(bitArray.Length); // 64 bit, every int is 32 bit

var bits = new BitArray(2);
bits[1] = true;

bits.Xor(bits);                 // Bitwise exclusive-OR bits with itself
Console.WriteLine(bits[1]);     // False
```

HashSet<T> and SortedSet<T>



HashSet<T> and SortedSet<T> ویژگی‌های متمایز زیر را دارند:

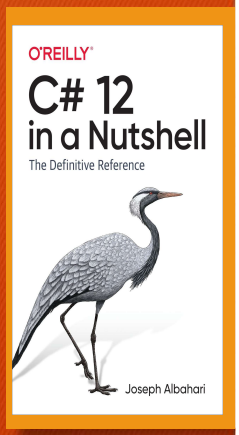
- متد Contains با استفاده از جستجوی مبتنی بر hash «hash-based» با سرعت بالایی اجرا میشه.
- اونها مقادیر تکراری رو نگه نمیدارند و در صورتی که بخواییم مقدار تکراری بهش اضافه کنیم اونو ignore میکنه.
- نمیتونید به یه عنصر براساس موقعیتش دسترسی داشته باشید.

SortedSet<T> عناصر رو به صورت مرتب نگه میداره ولی HashSet<T> اینکار رو نمیکنه.

هر دو کالکشن از اینترفیس ISet<T> ارث بردند. از .Net 5 به بعد این هر دو کلاس اینترفیس IReadOnlySet<T> را نیز پیاده‌سازی کردند.

```
HashSet<int> set = new HashSet<int>();
set.Add(10);
set.Add(20);
set.Add(20);
Console.WriteLine(set.Count);  // 2

SortedSet<string> sortedStrings = new SortedSet<string>();
sortedStrings.Add("a");
sortedStrings.Add("c");
sortedStrings.Add("a");
sortedStrings.Add("b");
Console.WriteLine(sortedStrings.Count);  // 3
```

HashSet<T> and SortedSet<T>



HashSet<T> با یک hashtable که فقط کلیدها رو ذخیره میکنه پیاده‌سازی میشه.

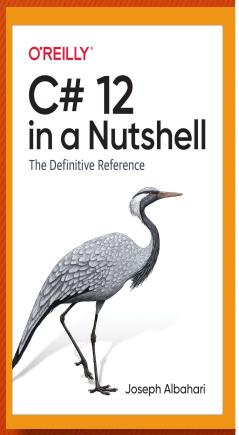
SortedSet<T> با یک red/black tree پیاده‌سازی میشه.

هر دوی این کالکشن‌ها اینترفیس ICollection<T> را پیاده‌سازی میکنند و متدهایی مثل Contain، Add و Remove را در اختیار داریم. همچنین متدی برای حذف مبتنی بر شرط به نام RemoveWhere هم وجود داره.

مثال زیر نشون میده چطور یک HashSet<char> جدید از روی کالکشن موجود میسازیم، همونطور که میبینید عناصر تکراری حذف شدند:

```
var letters = new HashSet<char>("the quick brown fox");

Console.WriteLine(letters.Contains('t'));      // true
Console.WriteLine(letters.Contains('j'));      // false

foreach (char c in letters) Console.Write(c);   // the quickbrownfx
```

HashSet<T> and SortedSet<T>



متدهای جذابی برای عملیات‌های مختلف با این کالکشن‌ها وجود داره، چهار متد اول destructive هستند چرا که عناصر یک set را دچار تغییر میکنند و مابقی متدها nondestructive هستند:

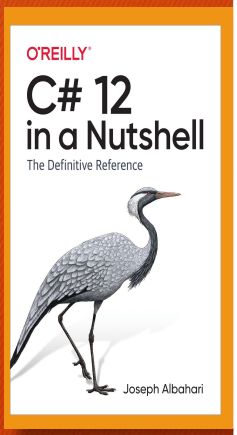
```
public void UnionWith(IEnumerable<T> other)
public void IntersectWith(IEnumerable<T> other)
public void ExceptWith(IEnumerable<T> other)
public void SymmetricExceptWith(IEnumerable<T> other)
public bool IsSubsetOf(IEnumerable<T> other)
public bool IsProperSubsetOf(IEnumerable<T> other)
public bool IsSupersetOf(IEnumerable<T> other)
public bool IsProperSupersetOf(IEnumerable<T> other)
public bool Overlaps(IEnumerable<T> other)
public bool SetEquals(IEnumerable<T> other)
```

UnionWith همه عناصر set دوم را به set اصلی اضافه میکنه.

IntersectWith همه عناصری که در هردو set وجود نداره رو حذف میکنه، در مثال زیر همه حروف مصوت را استخراج میکنیم:

```
letters = new HashSet<char>("the quick brown fox");
letters.IntersectWith("aeiou");
foreach (char c in letters) Console.Write(c);     // euio
```

HashSet<T> and SortedSet<T>



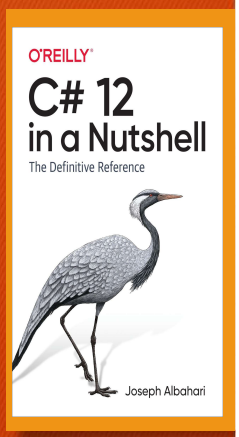
ExceptWith همه عناصر set دوم را از set اصلی حذف میکنه، در مثال زیر همه حروف مصوت رو حذف کردیم:

```
letters = new HashSet<char>("the quick brown fox");
letters.ExceptWith("aeiou");
foreach (char c in letters) Console.Write(c);     // th qckbrwnfx
```

SymmetricExceptWith همه عناصر مشترک بین دو set رو حذف میکنه و فقط عناصری که در یکی از اونها هستند رو نگه میداره:

```
letters = new HashSet<char>("the quick brown fox");
letters.SymmetricExceptWith("the lazy brown fox");
Console.WriteLine();
foreach (char c in letters) Console.Write(c);     // quicklazy
```

HashSet<T> and SortedSet<T>



SortedSet<T> همه متدهای HashSet<T> رو داره به اضافه یک سری متد اضافه زیر:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

اینم یه مثال از SortedList<T>

```
var sortedLetters = new SortedSet<char>("the quick brown fox");
foreach (char c in sortedLetters) Console.Write(c);   //  bcefhiknoqrtuwx

foreach (char c in sortedLetters.GetViewBetween('f', 'i'))
    Console.Write(c);                                 //  fhi
```