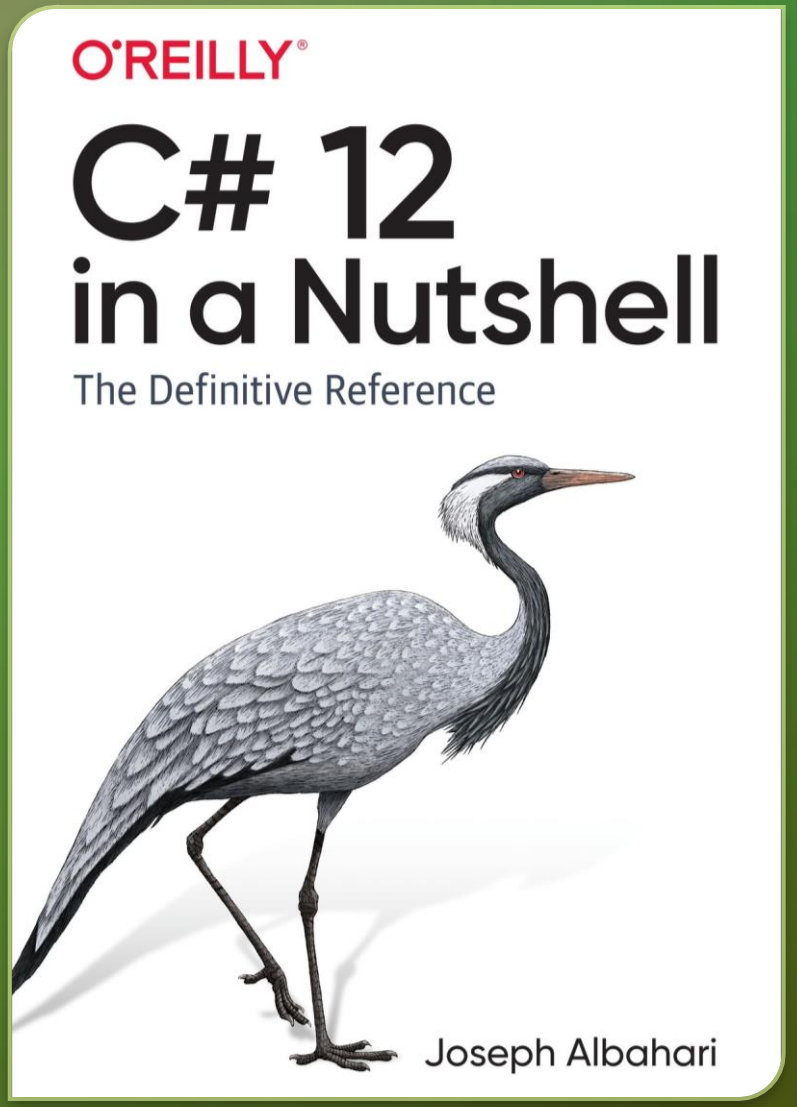
Chapter 7 - . Collections

- Enumeration
- IEnumerable and Ienumerator
- IEnumerable<T> and IEnumerator<T>
- Implementing the Enumeration Interfaces
- The ICollection and IList Interfaces
- ICollection<T> and Icollection
- IList<T> and Ilist
- IReadOnlyCollection<T> and IReadOnlyList<T>
- The Array Class



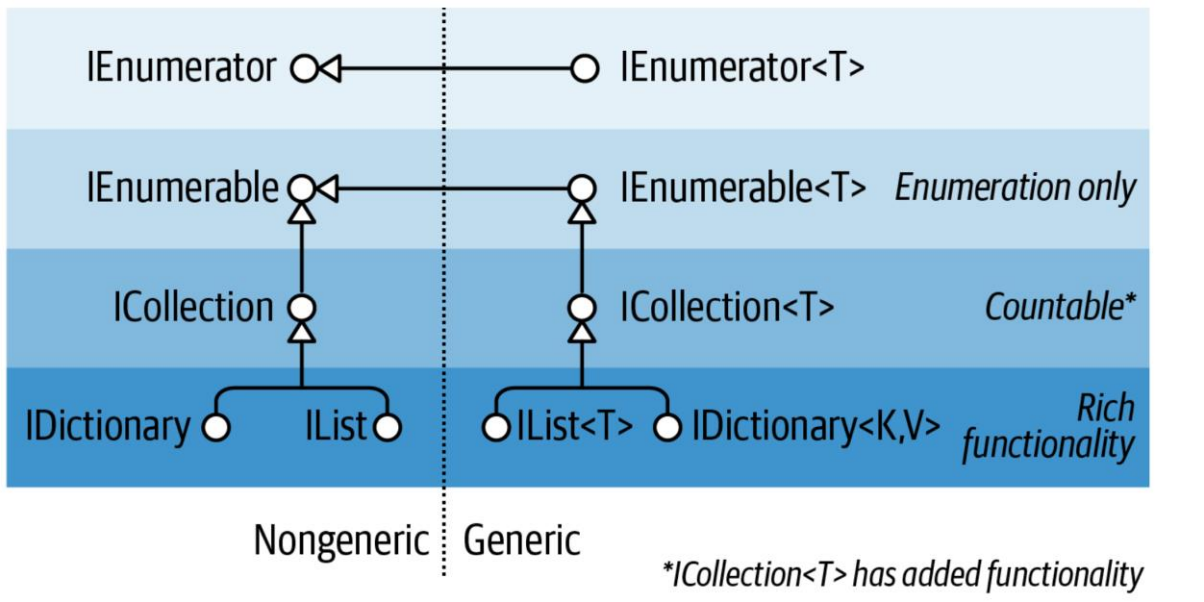
Collections

دات نت برای کار با کالکشن ها typeهای متنوعی رو معرفی کرده. میتونیم این انواع رو به سه دسته زیر تقسیم کنیم:

- Interfaces that define standard collection protocols
- Ready-to-use collection classes (lists, dictionaries, etc.)
- Base classes for writing application-specific collections

Namespace	Contains
`System.Collections`	Nongeneric collection classes and interfaces
`System.Collections.Specialized`	Strongly typed nongeneric collection classes
`System.Collections.Generic`	Generic collection classes and interfaces
`System.Collections.ObjectModel`	Proxies and bases for custom collections
`System.Collections.Concurrent`	Thread-safe collections (see Chapter 23)

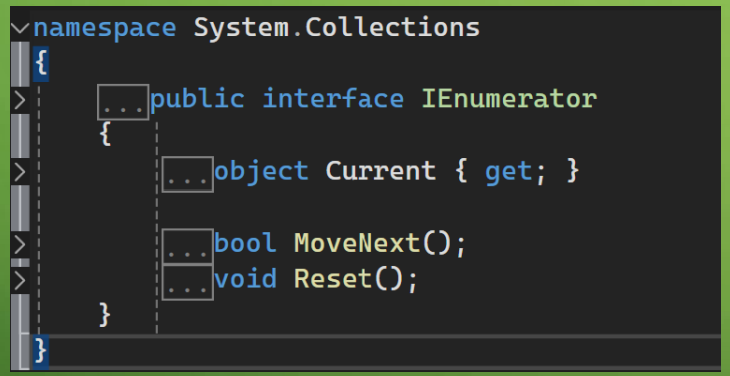
Enumeration



سی شارپ انواع متفاوتی از کالکشن‌ها رو معرفی می‌کنه. از کالکشن های ساده مثل آرایه و linkedList ها تا ساختارهای پیچیده تر مثل red/black trees و hashtables.
اگرچه این ساختارها پیاده‌سازی های گسترده ای دارند، توانایی پیمایش این ساختارها یک نیاز عمومی هستش. BCL در دات‌نت این امکان رو با اینترفیس‌هایی که در اختیارمون گذاشته برطرف میکنه.

IEnumerable and IEnumerator

اینترفیس IEnumerator یک کالکشن در پایین ترین سطح تعریف میکنه که آیتمهاش فقط روبه جلو میتونند پیمایش یا شمارش بشن. این اینترفیس به این شکل در فضای نام system.collections تعریف شده است.



MoveNext آیتم جاری (current) رو به آیتم بعدی منتقل میکنه. اگه آیتم بعدی وجود نداشته باشه false برمیگردونه.

Current آیتم جاری رو برمیگردونه و معمولا از نوع object به انواع دیگه تبدیل میشه. قبل از اینکه بخواهید اولین آیتم رو بگیرید، باید moveNext رو صدا بزنید.

Reset پوزیشن آیتم جاری رو به حالت اول برمیگردونه.

IEnumerable and IEnumerator

```
namespace System.Collections
{
    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }
}
```

کالکشن‌ها معمولا enumeraorها رو پیاده سازی نمیکنند و به جاش از طریق اینترفیس IEnumerable اون رو تامین میکنند.

این مثال رو ببینید:

```
string s = "Hello";

// Because string implements IEnumerable, we can call GetEnumerator():
IEnumerator rator = s.GetEnumerator();

while (rator.MoveNext())
{
    char c = (char)rator.Current;
    Console.Write(c + ".");
}

// Output:  H.e.l.l.o.
```

- کلاس string اینترفیس IEnumerable رو پیاده سازی کرده؛ برای همین میتونیم از متد GetEnumerator استفاده کنیم.

IEnumerable<T> and IEnumerator<T>

دو اینترفیسی که در اسلایدهای قبلی معرفی شد به صورت جنریک هم توسعه داده شدند:

```
namespace System.Collections.Generic
{
    public interface IEnumerator<out T> : IEnumerator, IDisposable
    {
        T Current { get; }
    }
}
```

```
namespace System.Collections.Generic
{
    public interface IEnumerable<out T> : IEnumerable
    {
        IEnumerator<T> GetEnumerator();
    }
}
```

Implementing the Enumeration Interfaces

شما ممکنه به یکی از دلایل زیر نیاز داشته باشید که IEnumerable رو پیاده سازی کنید:

- To support the foreach statement
- To interoperate with anything expecting a standard collection
- To meet the requirements of a more sophisticated collection interface
- To support collection initializers

Implementing the Enumeration Interfaces

برای پیاده سازی IEnumerable/IEnumerable<T> ما باید یک enumerator داشته باشیم.
اینکار رو به یکی از این سه روش میتونیم انجام بدیم:

- If the class is “wrapping” another collection, by returning the wrapped collection’s enumerator
- Via an iterator using yield return
- By instantiating your own IEnumerator/IEnumerator<T> implementation

Implementing the Enumeration Interfaces

```
0 references | 0 changes | 0 authors, 0 changes
public class MyCollection : IEnumerable
{
    int[] data = { 1, 2, 3 };

    0 references | 0 changes | 0 authors, 0 changes
    public IEnumerator GetEnumerator()
    {
        foreach (int i in data)
            yield return i;
    }
}

0 references | 0 changes | 0 authors, 0 changes
public class MyGenCollection : IEnumerable<int>
{
    int[] data = { 1, 2, 3 };

    1 reference | 0 changes | 0 authors, 0 changes
    public IEnumerator<int> GetEnumerator()
    {
        foreach (int i in data)
            yield return i;
    }

    // Explicit implementation keeps it hidden:
    0 references | 0 changes | 0 authors, 0 changes
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

این مثال رو ببینید:

برای پیاده سازی متد GetEnumerable از yield استفاده کردیم.

همینطور برای پیاده سازی کلاس جنریک، از اونجایی که این کلاس از کلاس غیرجنریک مشتق شده مجبوریم تا هر دو حالت متد رو پیاده سازی کنیم.

The ICollection and IList Interfaces

اگرچه اینترفیس‌های enumeration برای پیمایش کالکشن‌ها یک پروتکل فراهم میکنن، ولی نمیتونند مکانیزمی برای تخمین سایز کالکشن یا دسترسی به یک عنصر خاص از کالکشن یا ویرایش اون رو برامون فراهم کنه.

دات‌نت اینترفیس‌های ICollection, IList, و Idictionary رو معرفی کرده که هر سه هم به صورت جنریک هم غیرجنریک وجود دارد. هرچند مدل غیرجنریک بیشتر برای پشتیبانی در کدهای قدیمی وجود دارد.

ICollection<T> and ICollection

ICollection<T> یک اینترفیس استاندارد برای کالکشن‌های countable هست. این اینترفیس امکان تعیین تعداد اعضای یک کالکشن، جستجوی یک آیتم در کالکشن، کپی کالکشن در یک آرایه و تشخیص اینکه کالکشن read-only هست یا نه رو فراهم میکنه. برای کالکشن‌های writable میتونیم از متدهای Add، Remove و Clear استفاده کنیم.

همینطور به این علت که این اینترفیس از IEnumerable<T> مشتق شده، میتونیم با foreach اونو پیمایش کنیم.

ICollection<T> and ICollection

```
namespace System.Collections.Generic
{
    public interface ICollection<T> : IEnumerable<T>, IEnumerable
    {
        int Count { get; }
        bool IsReadOnly { get; }

        void Add(T item);
        void Clear();
        bool Contains(T item);
        void CopyTo(T[] array, int arrayIndex);
        bool Remove(T item);
    }
}
```

ICollection<T> and ICollection

```
namespace System.Collections
{
    public interface ICollection : IEnumerable
    {
        int Count { get; }
        bool IsSynchronized { get; }
        object SyncRoot { get; }

        void CopyTo(Array array, int index);
    }
}
```

مدل غیر جنریک ICollection هم کالکشن‌های countable رو میتونه برامون تعریف کنه با این تفاوت که متدی برای ویرایش کالکشن یا جستجو در اون رو نداره.

این اینترفیس‌ها معمولا توسط اینترفیس‌های IList و IDictionary پیاده سازی میشوند.

IList<T> and IList

IList<T> یک اینترفیس استاندارد برای کار با کالکشن‌های indexable هست. همچنین تمامی قابلیت‌های ICollection<T> و IEnumerable<T> رو هم به ارث میبره. همچنین قابلیت خوندن یا نوشتن در کالکشن براساس موقعیت یک آیتم را فراهم میکنه.

```
namespace System.Collections.Generic
{
    public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
    {
        T this[int index] { get; set; }

        int IndexOf(T item);
        void Insert(int index, T item);
        void RemoveAt(int index);
    }
}
```

متد indexOf یک جستجوی linear را روی لیست انجام میده و اگر آیتمی پیدا نکنه ۱- رو برمیگردونه (اگر هم پیدا کنه ایندکس آیتم رو برمیگردونه)

IList<T> and IList

در نسخه غیرجنریک اعضای بیشتری داریم چرا که چیزهای کمتری از ICollection به ارث میبره:

```
namespace System.Collections
{
    public interface IList : ICollection, IEnumerable
    {
        object? this[int index] ...
        bool IsFixedSize { get; }
        bool IsReadOnly { get; }

        int Add(object? value);
        void Clear();
        bool Contains(object? value);
        int IndexOf(object? value);
        void Insert(int index, object? value);
        void Remove(object? value);
        void RemoveAt(int index);
    }
}
```

متد Add اینجا ایندکس آیتمی که به کالکشن اضافه شده رو برمیگردونه. در صورتی که این متد در اینترفیس ICollection<T> خروجیش void هست.

کلاس List<T> و Array هر دوی این اینترفیس‌ها رو پیاده سازی کردند.

IReadOnlyCollection<T> and IReadOnlyList<T>

دات نت برای کار با کالکشن‌های صرفا read-only دو اینترفیس اختصاصی معرفی کرده.

```
namespace System.Collections.Generic
{
    public interface IReadOnlyCollection<out T> : IEnumerable<T>, IEnumerable
    {
        int Count { get; }
    }
}
```

```
namespace System.Collections.Generic
{
    public interface IReadOnlyList<out T> : IEnumerable<T>, IEnumerable, IReadOnlyCollection<T>
    {
        T this[int index] { get; }
    }
}
```

The Array Class

کلاس Array یک کلاس پایه برای همه انواع آرایه های یک یا چند بعدی است. این کلاس یکی از مهمترین پیاده‌سازی های اینترفیس‌های کالکشن است که یک الگوی واحد رو برامون فراهم میکنه، بنابراین بدون توجه به نوع آرایه تعریف شده یک سری متدهای عمومی جهت کار با آن همیشه در دسترس داریم.

ازونجایی که Array خیلی پایه‌ای و مهمه، سی‌شارپ برای تعریف و مقداردهی آن syntax خاصی رو فراهم کرده.

Array اینترفیس‌های کالکشن رو تا IList<T> در هردو فرمت جنریک و غیرجنریک پیاده سازی میکنه، متدهای مختص لیست مثل Add یا Remove در کلاس Array به صورت NotSupport پیاده‌سازی شدند.

The Array Class

```
int IList.Add(object? value)
{
    ThrowHelper.ThrowNotSupportedException(ExceptionResource.NotSupported_FixedSizeCollection);
    return default;
}

bool IList.Contains(object? value)
{
    return IndexOf(this, value) >= this.GetLowerBound(0);
}

void IList.Clear()
{
    Clear(this);
}

int IList.IndexOf(object? value)
{
    return IndexOf(this, value);
}

void IList.Insert(int index, object? value)
{
    ThrowHelper.ThrowNotSupportedException(ExceptionResource.NotSupported_FixedSizeCollection);
}

void IList.Remove(object? value)
{
    ThrowHelper.ThrowNotSupportedException(ExceptionResource.NotSupported_FixedSizeCollection);
}

void IList.RemoveAt(int index)
{
    ThrowHelper.ThrowNotSupportedException(ExceptionResource.NotSupported_FixedSizeCollection);
}
```

اینجا رو ببینید:

این بخشی از پیاده‌سازی اینترفیس IList توسط کلاس Array هست.

برای تغییر سایز آرایه، یک متد مجزا به نام Resize تعبیه شده که در اصل یک آرایه جدید با سایز مشخص شده رو میسازه.

The Array Class

عناصر یک آرایه میتونند reference-type یا value-type باشند.

عناصر value-type همونجایی که آرایه تعریف شده ذخیره میشن، مثلا آرایه‌ای با سه عنصر long (هر کدوم ۸ بایت) ۲٤ بایت از حافظه رو اشغال میکنند.

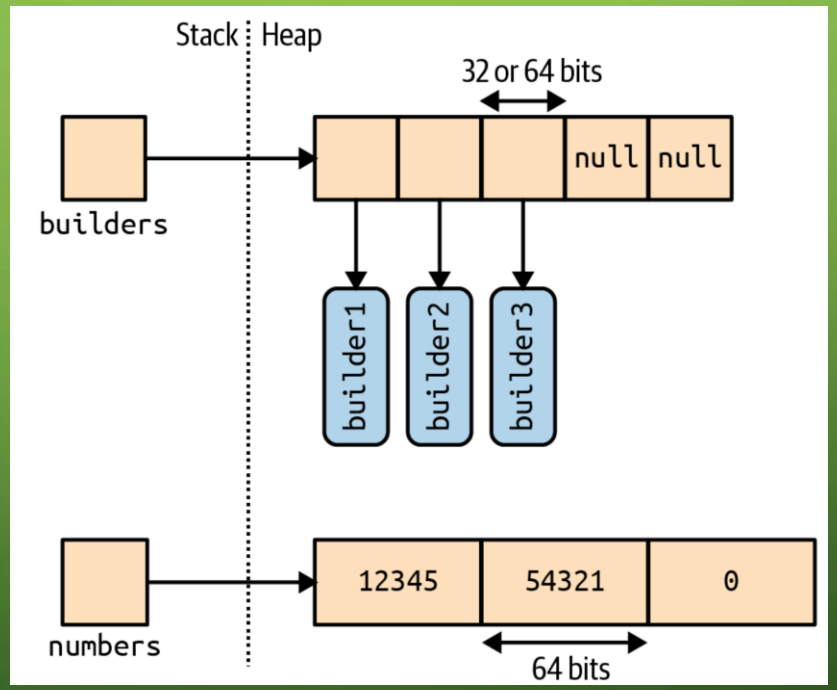
اما عناصر reference-type فقط به اندازه مورد نیازشون فضا اشغال میکنند.

```
StringBuilder[] builders = new StringBuilder[5];
builders[0] = new StringBuilder("builder1");
builders[1] = new StringBuilder("builder2");
builders[2] = new StringBuilder("builder3");

long[] numbers = new long[3];
numbers[0] = 12345;
numbers[1] = 54321;
```

The Array Class

این تصویر نحوه ذخیره و فضای اشغالی در حافظه رو برای کد صفحه قبل نشون میده:



The Array Class

ازونجایی که Array یک کلاس هست پس همیشه reference-type است. (بدون در نظر گرفتن نوع عناصرش). این یعنی اگر ما بنویسیم: arrayA = arrayB در نتیجه دو متغیر داریم که یک رفرنس دارند.

به همین ترتیب دو آرایه مجزا هیچ وقت باهم برابر نیستند مگر اینکه از مقایسه structural equality استفاده کنید.

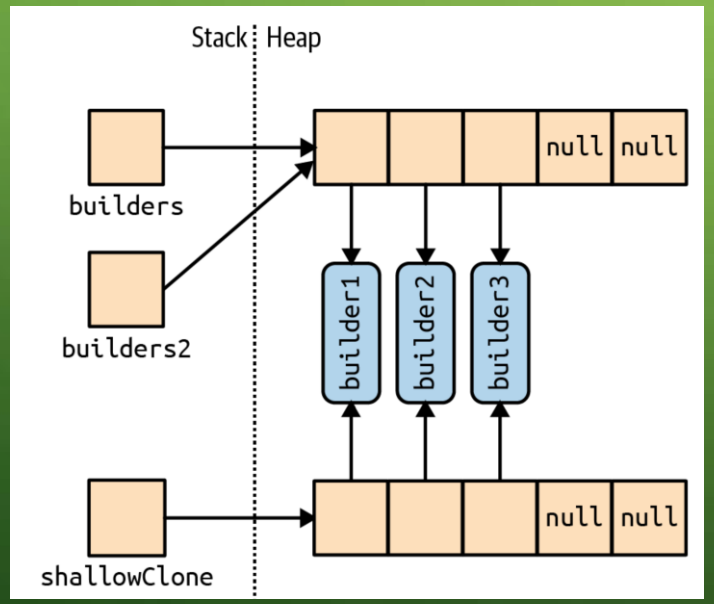
```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

Console.WriteLine(a1 == a2);                              // False
Console.WriteLine(a1.Equals(a2));                         // False

IStructuralEquatable se1 = a1;
Console.WriteLine(se1.Equals(a2,
 StructuralComparisons.StructuralEqualityComparer));   // True
```

The Array Class

آرایه ها میتونند با استفاده از متد Clone کپی بشن. البته که این متد صرفا یک shallow-copy ایجاد میکنه، یعنی صرفا محل ذخیره آرایه در حافظه کپی میشه. اگر عناصر آرایه value-type باشند، خودشون کپی میشن ولی اگر reference-type باشند صرفا رفرنس اونها کپی میشه و عملا دو آرایه داریم که به یک نقطه از حافظه اشاره میکنند.




```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[])builders.Clone();
```

The Array Class

برای ایجاد deep copy از یک آرایه با عناصر reference-type نیازه که روی تمامی آیتمهای آن پیمایش کنید و از همه عناصر clone بگیرید.