Generic Collection Interfaces

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/

Interface

A specification identifying a related set of properties and methods.

A class commits to supporting the specification by implementing the interface.

Use the interface as a data type to work with any class that implements the interface.

Interface Is a Specification

```
public interface ICollection<T> : IEnumerable<T>
{
    int Count { get; }
    void Add(T item);
    void Clear();
    bool Contains(T item);
    bool Remove(T item);
}
```

Implementing an Interface

```
public class List<T> : ICollection<T>
{
    private T[] _items;
    private int _size;
    public int Count
    {
        get
        {
            return _size;
        }
    }
    public void Add(T item)
    {
        if (_size == _items.Length) EnsureCapacity(_size + 1);
        _items[_size++] = item;
    }
    . . .
}
```

ICollection<T> interface

```
int Count { get; }
void Add(T item);
void Clear();
bool Contains(T item);
bool Remove(T item);
```

Using an Interface as a Data Type

```
public string SendEmail(ICollection<Vendor> vendors, string message)
{
    var confirmation = "";
    var emailService = new EmailService();
    Console.WriteLine(vendors.Count);
    foreach (var vendor in vendors)
    {
        var subject = "Important message for: " + vendor.CompanyName;
        confirmation += emailService.SendMessage(subject,
                                                 message,
                                                 vendor.Email);
    }
    return confirmation;
}
```

ICollection<T> interface

```
int Count { get; }
void Add(T item);
void Clear();
bool Contains(T item);
bool Remove(T item);
```

C# Interfaces

Built-in interfaces

Generic collection interfaces

Custom interfaces

"Object-Oriented Programming Fundamentals in C#"

Overview

Making the Case for Using Interfaces

Built-In Generic Collection Interfaces

Using an Interface as a Parameter

Using an Interface as a Return Type

Returning IEnumerable<T>

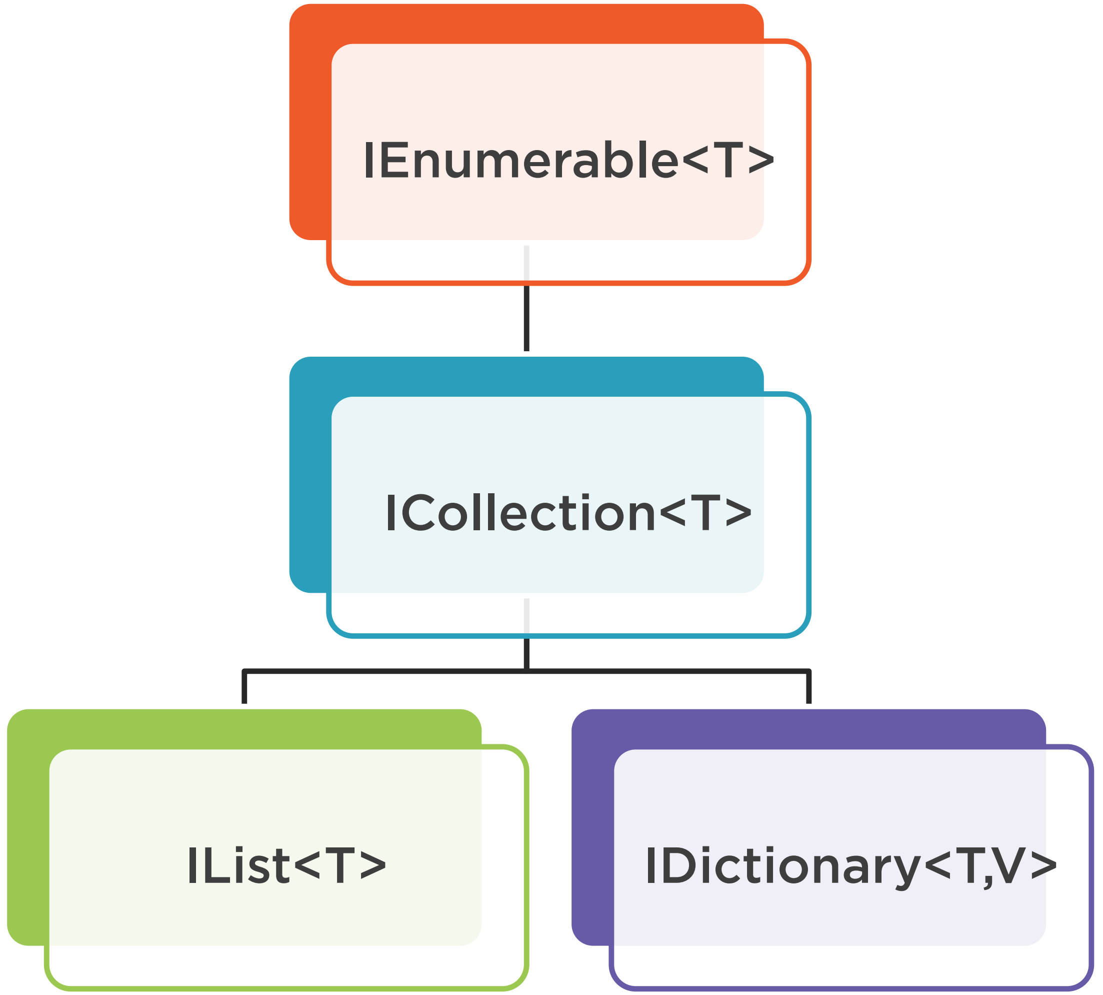
Defining an Iterator with `yield`

FAQ

Demo

The case for interfaces

Generic Collection Interfaces



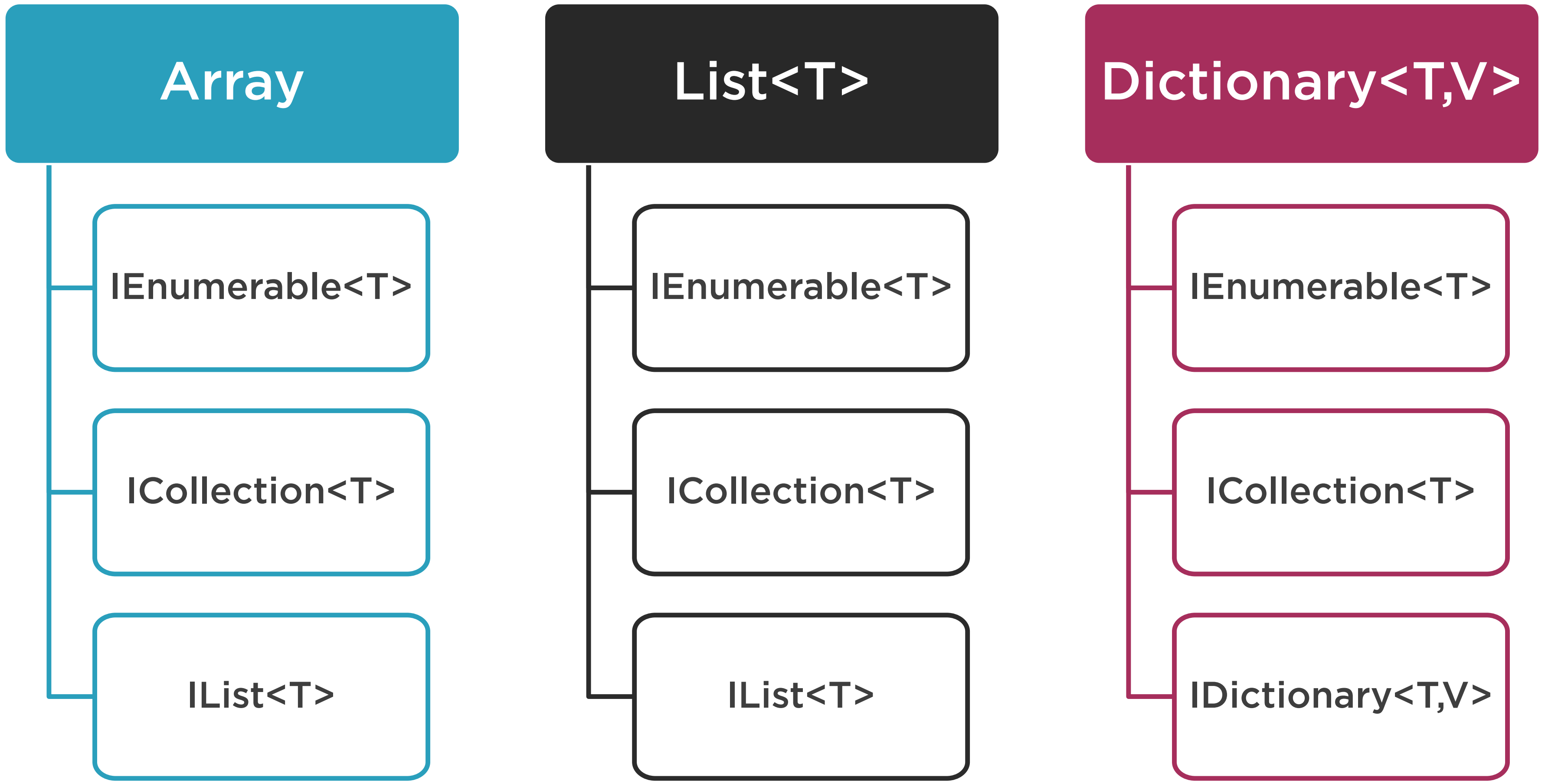
GetEnumerator

Count
Add
Remove

IndexOf
Insert
RemoveAt

Keys
Values
TryGetValue

Generic Collection Interfaces



Using an Interface

Parameter

The calling code can pass in an instance of any collection class that implements the interface

Return Type

The calling code can cast the returned value to any collection class that implements the interface

Class

A custom collection class implements the interface

Using an Interface as a Parameter

```
public List<string> SendEmail(List<Vendor> vendors, string message)
{

}

public List<string> SendEmail(ICollection<Vendor> vendors, string message)
{

}
```

Interface as a Parameter Best Practices

Do:

Consider using an interface instead of a concrete type when passing collections to methods

Avoid:

Using any of the interfaces in the System.Collections namespace

Using IEnumerable<T> as the parameter data type
Unless the method simply iterates through the collection

Using an Interface

Parameter

The calling code can pass in an instance of any collection class that implements the interface

Return Type

Using an Interface as a Return Type

```
public List<Vendor> Retrieve()
{

}


public ICollection<Vendor> Retrieve()
{


}
```

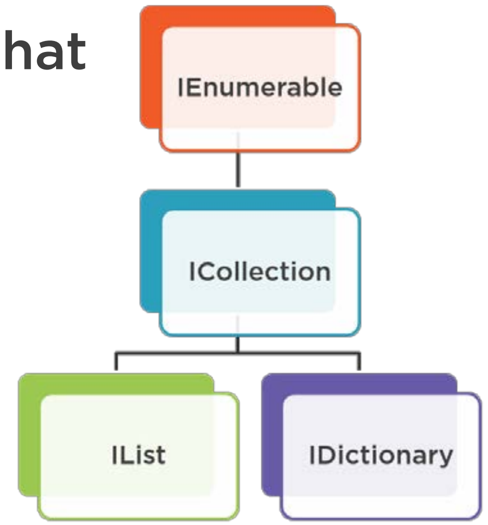
Interface as a Return Type Best Practices

Do:

Consider using an interface when returning a collection from a method

Use a cast operation to cast the result to the desired collection type

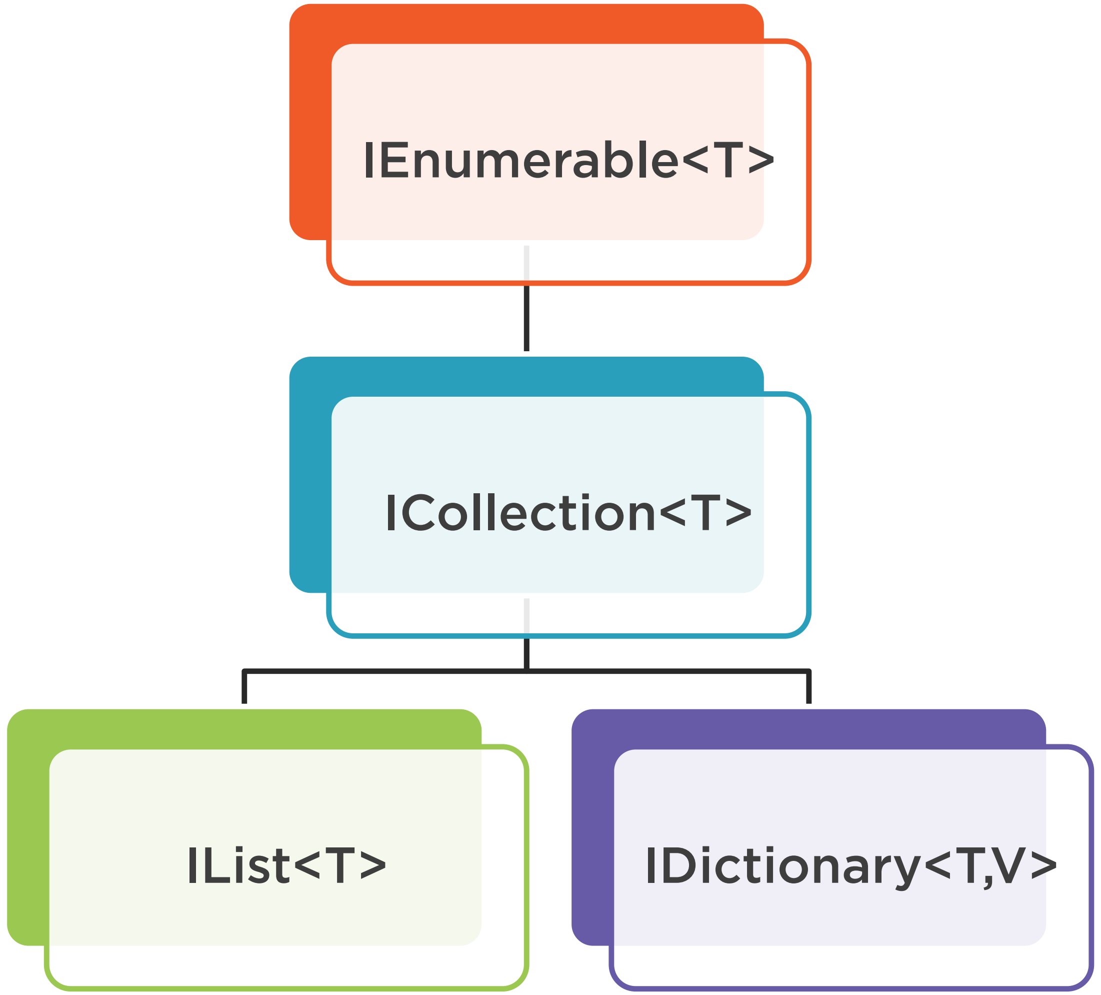
Use the most general interface that meets the requirements



Avoid:

Using any of the interfaces in the System.Collections namespace

IEnumerable<T>



GetEnumerator

Count
Add
Remove

IndexOf
Insert
RemoveAt

Keys
Values
TryGetValue

Demo

The case for returning IEnumerable<T>

Returning IEnumerable<T> Best Practices

Do:

Consider returning an IEnumerable<T> to provide an immutable collection

Consider returning IEnumerable<T> when the calling code use cases are unknown

Consider returning IQueryable<T> when working with a query provider, such as LINQ to SQL or Entity Framework

Avoid:

Returning an IEnumerable<T> if the collection must be modified by the caller

Returning an IEnumerable<T> if the caller requires information about the collection, such as the count

Returning an IEnumerable<T> if the caller must be notified of a change to the collection

Defining an Iterator with **`yield`**

```
public IEnumerable<int> Fibonacci(int x)
{
    int prev = -1;
    int next = 1;
    for (int i = 0; i < x; i++)
    {
        int sum = prev + next;
        prev = next;
        next = sum;
        yield return sum;
    }
}
```

Iterator Best Practices

Do:

Use an iterator when a method should return one element at a time
Lazy Evaluation

Use an iterator for deferred execution

Avoid:

Using an iterator if it's not required

Frequently Asked Questions

- What is an interface?
  - A specification for identifying a related set of properties and methods.
- What does it mean to say that a class implements an interface?
  - A class commits to supporting the specification defined in the interface by implementing code for each property and method.
- What is a key benefit of using an interface as a data type?
  - We can write more generalized code when defining properties, method parameters, or method return values.

Frequently Asked Questions (cont)

- **What does the IEnumerable<T> interface provide?**
  - The ability to iterate through a collection using foreach for example.
- **What does the ICollection<T> interface provide?**
  - The ability to work with a collection: add or remove elements and get the element count for example.
- **What does the IList<T> interface provide?**
  - The ability to work with a list by index: locate items by index or insert at a specific index for example.

Summary

Making the Case for Using Interfaces

Built-In Generic Collection Interfaces

Using an Interface as a Parameter

Using an Interface as a Return Type

Returning IEnumerable<T>

Defining an Iterator with `yield`

Passing a Collection to a Method

Returning a Collection from a Method

IEnumerable<T>

Read-only sequence of elements

IList<T> or ICollection<T>

Flexible updatable collection

List<T>, Array[] or Dictionary<T,V>

Specific updateable collection