C# Best Practices: Collections and Generics

INTRODUCTION

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/

C# Best Practices Series

C# Best Practices: Improving on the Basics

C# Best Practices: Collections and Generics

- Components
- Building good
  - Classes
  - Properties
  - Methods
- String handling
- New C# 6 features

An Apprentice

A learner

One that has ability, but has yet to achieve all knowledge

Someone who wants to know more

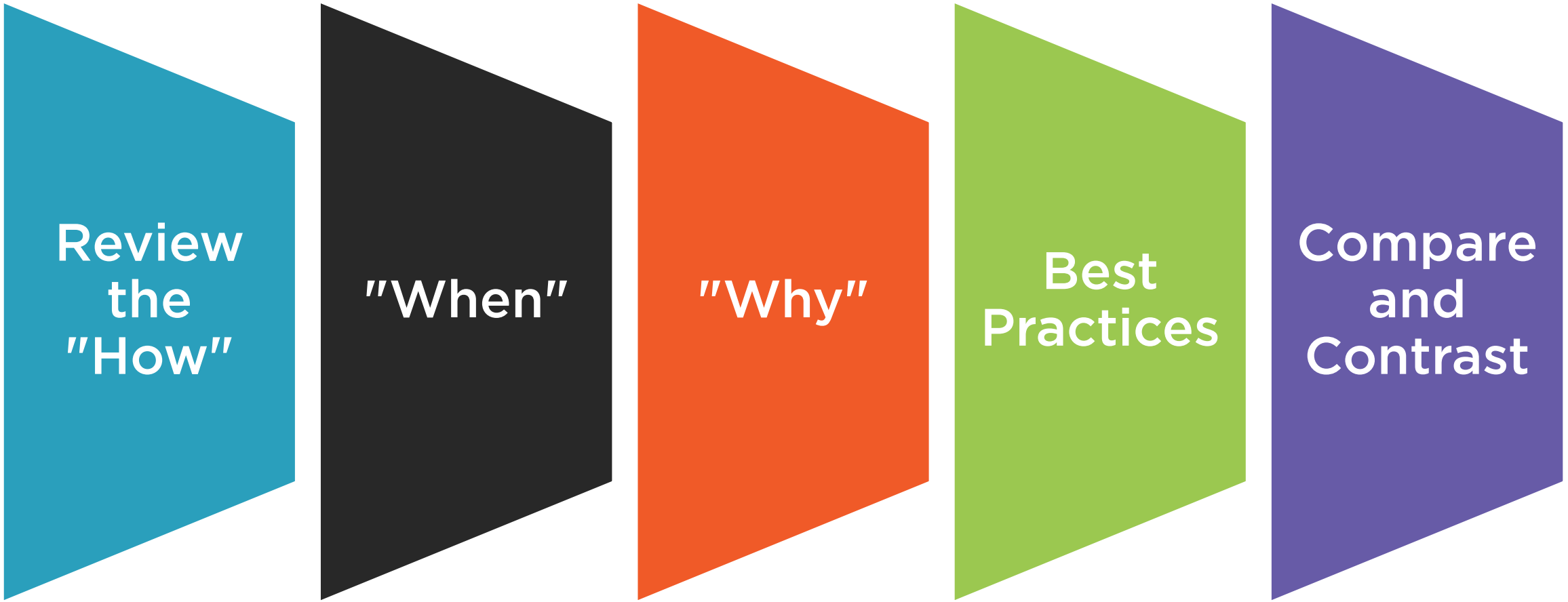
This Course Is for You!

Learn more about handing collections in your applications

Leverage generics to build more generalized code

Use LINQ to query your collections

For Each Covered Feature



Prerequisites

C#	**C# 6.0**
	C# 5.0
Visual Studio	**Visual Studio 2015**
	Enterprise Edition
	Professional Edition
	Community Edition
	Visual Studio 2013

https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx

Module Overview

Collection Overview

Get the most from this course

Sample Application

Course Outline

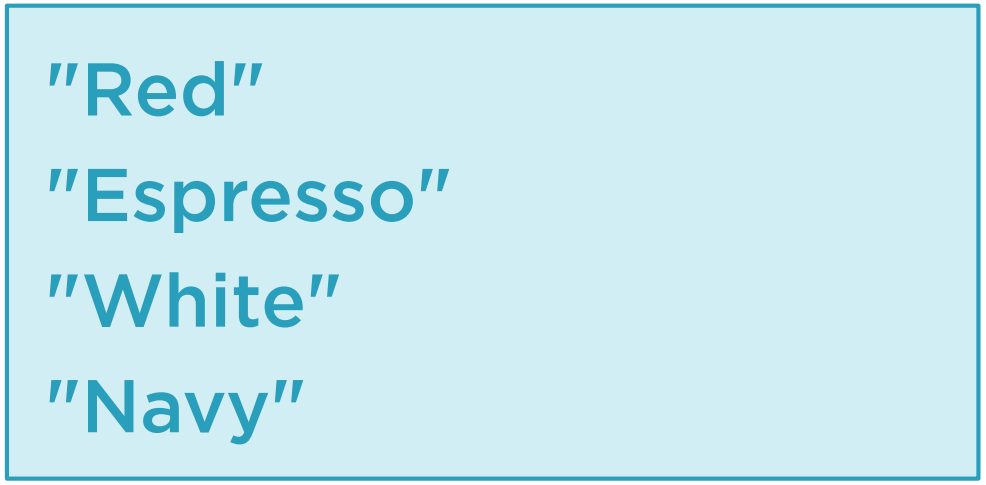
"Red"
"Espresso"
"White"
"Navy"

1 "Saw" 9.99
2 "Wrench" 8.98
3 "Steel Hammer" 15.95

Collection

A class that provides in-memory management of a group of items

- Simple types
- Complex types



Collection

A class that provides in-memory management of a group of items

- Similar

```
"Red"
1 "Saw" 9.99
button1
42
```

Collection

A class that provides in-memory management of a group of items

- Similar
- Different

Kinds of Collections

Lists

"Red"
"Espresso"
"White"
"Navy"

Dictionaries

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

List

0	"Red"
1	"Espresso"
2	"White"
3	"Navy"

Dictionary

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

Collection of What?

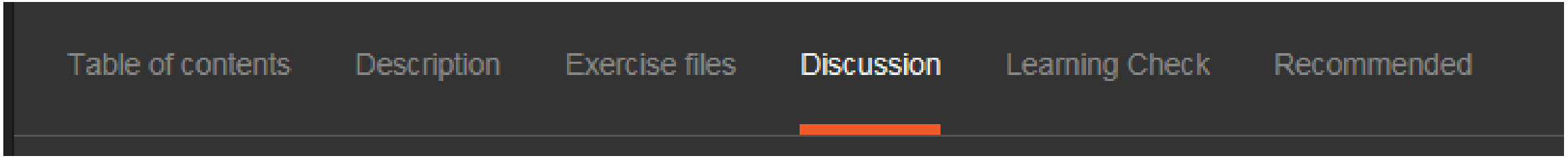
Product Objects

1 "Saw" 9.99
2 "Wrench" 8.98
3 "Steel Hammer" 15.95

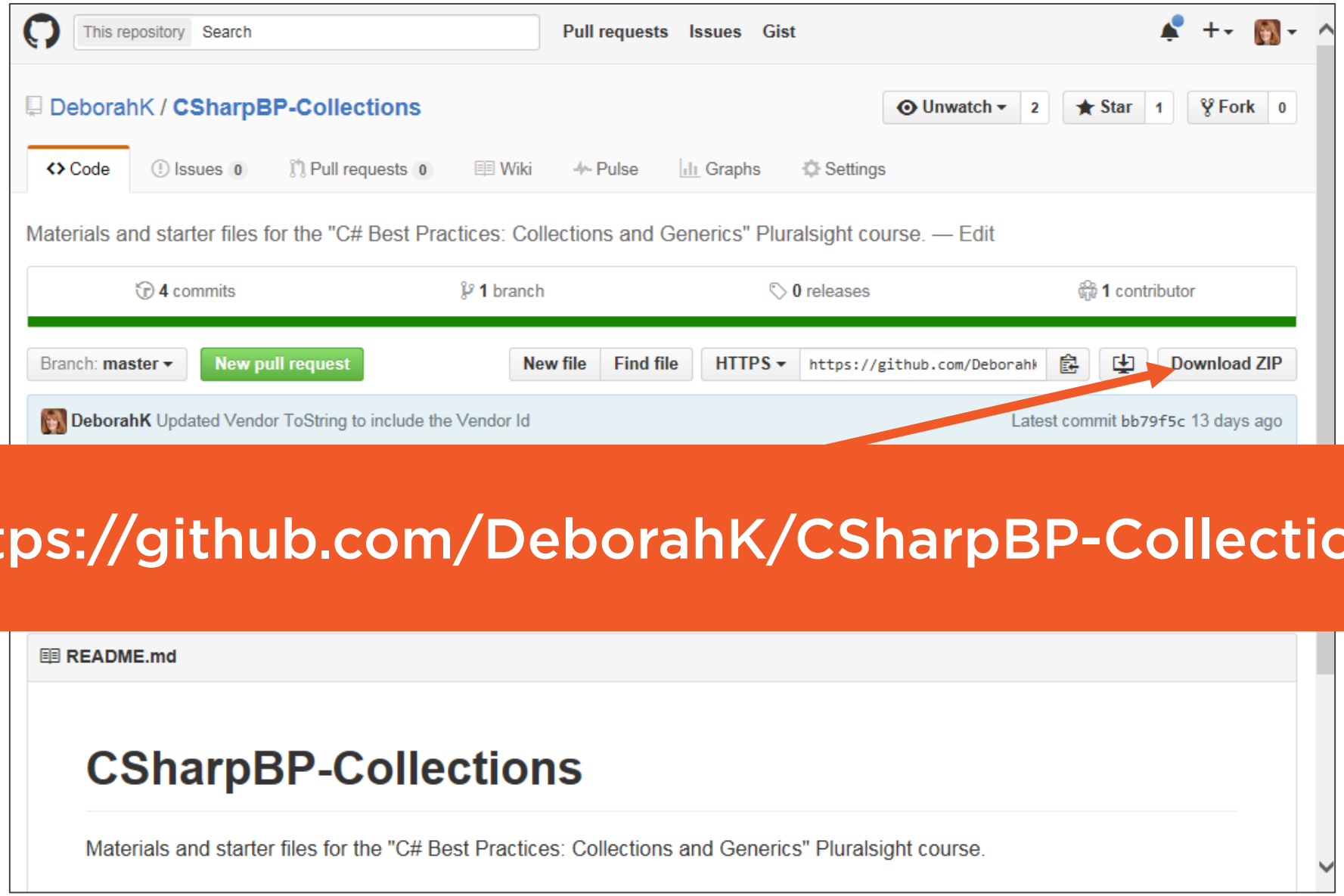
Strings

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

Thoughts? Comments? Questions?



GitHub Repository



Course Outline

- Arrays
- Building Generic Code with Generics
- Generic Lists
- Generic Dictionaries
- Generic Collection Interfaces
- LINQ
- Final Words

Arrays

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/

0	"Red"
1	"Espresso"
2	"White"
3	"Navy"

0	1 "Saw" 9.99
1	2 "Wrench" 8.98
2	3 "Steel Hammer" 15.95

Array

A fixed-size list of elements that can be accessed using a positional index number

Module Overview

Declaring and Populating an Array

Using Collection Initializers

Retrieving an Element from an Array

Iterating Through an Array

Using Array Methods

FAQ

Declaring an Array

```
string[] colorOptions;
```

- Collection of what?
- Array element type
- Array variable

"Red"
"Espresso"
"White"
"Navy"

Initializing an Array

```
string[] colorOptions;
colorOptions = new string[4];
```

- Reference type
- new keyword
- Type and size of the array

0	null
1	null
2	null
3	null

Declaring and Initializing an Array

```
string[] colorOptions;
colorOptions = new string[4];
```

```
string[] colorOptions = new string[4];
```

```
var colorOptions = new string[4];
```

Populating an Array

```
colorOptions[0] = "Red";
colorOptions[1] = "Espresso";
colorOptions[2] = "White";
colorOptions[3] = "Navy";
```

0	Red
1	Espresso
2	White
3	Navy

Array Best Practices

Do:

Consider using an array when the required size of a list can be determined at design time

Use a plural variable name for the array

Avoid:

Using an array when the size of the list is not known

Declaring and Populating an Array

```
var colorOptions = new string[4];
```

```
colorOptions[0] = "Red";
colorOptions[1] = "Espresso";
colorOptions[2] = "White";
colorOptions[3] = "Navy";
```

Collection Initializers

```
var colorOptions = new string[4];
```

```
colorOptions[0] = "Red";
colorOptions[1] = "Espresso";
colorOptions[2] = "White";
colorOptions[3] = "Navy";
```

```
string[] colorOptions = new string[4] {"Red", "Espresso", "White", "Navy"};
```

```
string[] colorOptions = {"Red", "Espresso", "White", "Navy"};
```

```
string[] colorOptions = {"Red", "Espresso", "White", GetMyFavoriteColor()};
```

Array Initialization Best Practices

Do:

Use collection initializers

Avoid:

Manually populating an array

Retrieving an Array Element

```
var colorOptions = new string[4];
```

```
colorOptions[0] = "Red";
colorOptions[1] = "Espresso";
colorOptions[2] = "White";
colorOptions[3] = "Navy";
```

```
Console.WriteLine(colorOptions[1]);
```

Retrieving Array Element Best Practices

Do:

Take care when referencing elements by index

Will generate a runtime exception

Avoid:

Retrieving elements by index when you need all elements

Iterate through instead

Iterating Through an Array

```
1 "Saw" 9.99
2 "Wrench" 8.98
3 "Steel Hammer" 15.95
```

```
"Red"
"Espresso"
"White"
"Navy"
```

`foreach`

`for`

Iterating an Array

`foreach`	`for`
Quick and easy	Complex but flexible
Iterate all elements	Iterate all or a subset of elements
Array element is read-only	Array element is editable

Using Array Methods

- Arrays derive from System.Array class

```
var brownIndex = Array.IndexOf(colorOptions, "Espresso");
```

```
colorOptions.SetValue("Blue", 3);
```

Frequently Asked Questions

- When is it appropriate to use an array?
  - When working with a list whose length is defined at design time.
  - When multiple dimensions are needed.
  - To squeeze out a bit more performance with large sets.
- What is the difference between `foreach` and `for` when iterating through an array?
  - `foreach` provides simple syntax for iterating all elements in an array.
  - `for` provides more complex but flexible syntax for iterating all or any subset of elements in an array.
    - Plus the iterated items are updateable.

Module Summary

- Declaring and Populating an Array
- Using Collection Initializers
- Retrieving an Element from an Array
- Iterating Through an Array
- Using Array Methods

Building Generic Code with Generics

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/

Generics Are ...

Writing code without specifying data types

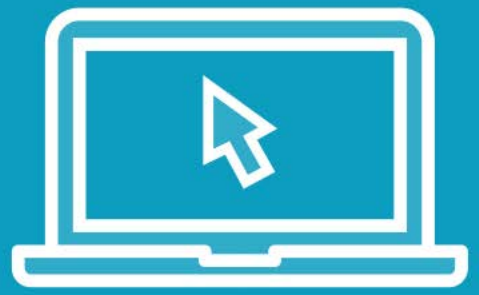
Yet type-safe

A way to make our code generic

Overview

- Making the case for generics
- Building a generic class
- Using a generic class
- Defining generic methods
- Leveraging generic constraints
- FAQ

Demo



The case for generics

One Class for Each Data Type

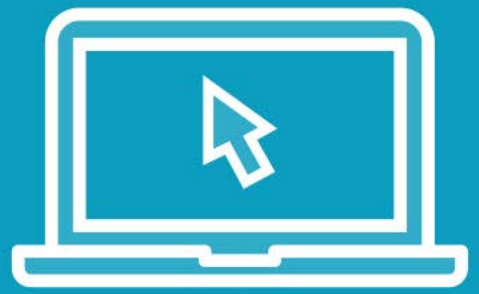
```
public class OperationResult
{
}

public class OperationResultDecimal
{
}

public class OperationResultInteger
{
}

public class OperationResultString
{
}
```

Demo



Building a generic class

```
public class OperationResult<T>
{
}
```

Generics ...

```
public class OperationResult<T>
{
    public OperationResult()
    {
    }

    public OperationResult(T result, string message) : this()
    {
        this.Result = result;
        this.Message = message;
    }

    public T Result { get; set; }
    public string Message { get; set; }
}
```

Multiple generic parameters

```
public class OperationResult<T, V>
{
    public OperationResult()
    {
    }

    public OperationResult(T result, V message) : this()
    {
        this.Result = result;
        this.Message = message;
    }

    public T Result { get; set; }
    public V Message { get; set; }
}
```

Generic Class Best Practices

Do:

Use generics to build reusable, type-neutral classes

Use T as the type parameter for classes with one type parameter

Prefix descriptive type parameter names with T

```
public class OpResult<TResult, TMessage>
```

Avoid:

Using generics when not needed

Using single-letter names when defining multiple type parameters

Use a descriptive name instead

Using a Generic Class

```
public class OperationResult<T>
{
    public OperationResult(){ }

    public OperationResult(T result, string message) : this()
    {
        this.Result = result;
        this.Message = message;
    }

    public T Result { get; set; }
    public string Message { get; set; }
}
```

```
var operationResult = new OperationResult<bool>(success, orderText);
```

```
var operationResult = new OperationResult<decimal>(value, orderText);
```

Defining Generic Methods

```
public int RetrieveValue(string sql, int defaultValue)
```

```
public T RetrieveValue(string sql, T defaultValue)
```

```
public class VendorRepository<T>
```

```
public T RetrieveValue<T>(string sql, T defaultValue)
```

Generic Method Best Practices

Do:

Use generics to build reusable, type-neutral methods

Use T as the type parameter for methods with one type parameter

Prefix descriptive type parameter names with T

```
public TReturn RetValue<TReturn, TParameter>
         (string sql, TParameter sqlParameter)
```

Define the type parameter(s) on the method signature

Avoid:

Using generics when not needed

Using single-letter names when defining multiple type parameters

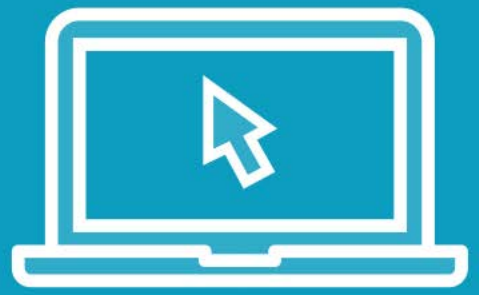
Use a descriptive name instead

Generic Method

```
public T RetrieveValue<T>(string sql, T defaultValue)
{
    // Call the database to retrieve the value
    // If no value is returned, return the default value
    T value = defaultValue;

    return value;
}
```

Demo



The Case for Generic Constraints

GENERIC CONSTRAINT	CONSTRAINS T TO
`where T : struct`	Value type
`where T : class`	Reference type
`where T : new()`	Type with parameterless constructor
`where T : Vendor`	Be or derive from Vendor
`where T : IVendor`	Be or implement the IVendor interface

Generic Constraint Syntax

```
public class OperationResult<T> where T : struct
```

```
public T Populate<T>(string sql) where T : class, new()
{
    T instance = new T();
    // Code here to populate an object
    return instance;
}
```

```
public T RetrieveValue<T, V>(string sql, V parameter)
                                        where T: struct
                                        where V: struct
```

Limits to Generic Constraints

```
public T RetrieveValue<T>(string sql, T defaultValue)
{
    // Call the database to retrieve the value
    // If no value is returned, return the default value
    T value = defaultValue;

    return value;
}
```

```
public string RetrieveValue(string sql, string defaultValue)
{
    // Call the database to retrieve the value
    // If no value is returned, return the default value
    string value = defaultValue;

    return value;
}
```

Frequently Asked Questions

- What are generics?
  - A technique for defining a data type using a variable.
- What are the benefits of generics?
  - With generics we can write generalized reusable code that is type-safe, yet works with any data type.

Frequently Asked Questions (cont)

- What is a generic type parameter?
  - A placeholder for the specific type
  - For example: `public class OperationResult<T>`
- Where is a generic type parameter defined?
  - As part of a class signature

```
public class OperationResult<T>
```

  - Or as part of a method signature

```
public T RetrieveValue<T>(string sql, T defaultValue)
```

Frequently Asked Questions (cont)

- In this example, how do you define the actual type for T?
  - ```
    public class OperationResult<T>
    ```
  - ```
    var operationResult = new OperationResult<decimal>();
    ```
  - ```
    var operationResult = new OperationResult<bool>();
    ```
- What is the purpose of a generic constraint?
  - To limit the types accepted for a generic type parameter.
    ```

Summary

- Making the case for generics
- Building a generic class
- Using a generic class
- Defining generic methods
- Leveraging generic constraints

Generic Lists

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/

Generic List

A strongly typed list of elements that is accessed using a positional index number

Array vs. Generic List

Array	Generic List
Strongly typed	Strongly typed
Fixed length	Expandable
No ability to add or remove elements	Can add, insert, or remove elements
Multi-dimensional	One-dimensional

Generic List

List<T>

List<int>

List<decimal>

List<string>

List<Product>

Overview

- Declaring and Populating a Generic List
- Using Collection Initializers
- Initializing a List of Objects
- Retrieving an Element from a Generic List
- Iterating Through a Generic List
- Types of C# Lists
- FAQ

Declaring a Generic List

```
List<string> colorOptions;
```

- List of what?
- List<T>
  - Where T is the type of elements the list contains

"Red"
"Espresso"
"White"
"Navy"

Initializing a Generic List

```
List<string> colorOptions;
colorOptions = new List<string>();
```

- Reference type
- new keyword

Declaring and Initializing a List

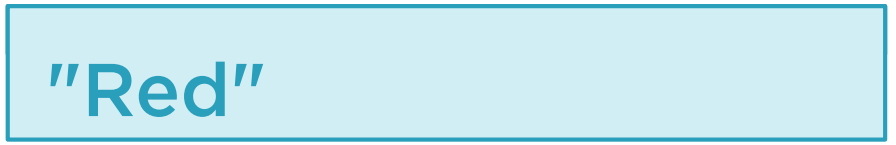
```
List<string> colorOptions;
colorOptions = new List<string>();
```

```
List<string> colorOptions = new List<string>();
```

```
var colorOptions = new List<string>();
```

Populating a List (Add)

```
colorOptions.Add("Red");
```



Populating a List (Add)

```
colorOptions.Add("Red");
colorOptions.Add("Espresso");
colorOptions.Add("White");
colorOptions.Add("Navy");
```

"Red"
"Espresso"
"White"
"Navy"

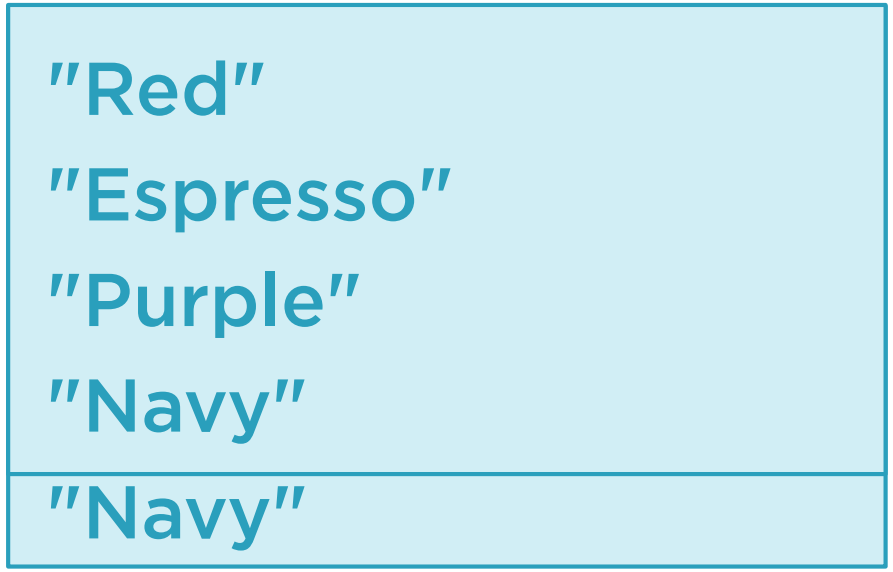
Populating a List (Insert)

```
colorOptions.Insert(2, "Purple");
```

"Red"
"Espresso"
"Purple"
"White"
"Navy"

Removing an Element

```
colorOptions.Remove("White");
```



Generic List Best Practices

Do:

Use generic lists to manage collections

Use Add over Insert where possible

Use a plural variable name for the list

Avoid:

Removing elements where possible

Declaring and Populating a List

```
var colorOptions = new List<string>();
```

```
colorOptions.Add("Red");
colorOptions.Add("Espresso");
colorOptions.Add("White");
colorOptions.Add("Navy");
```

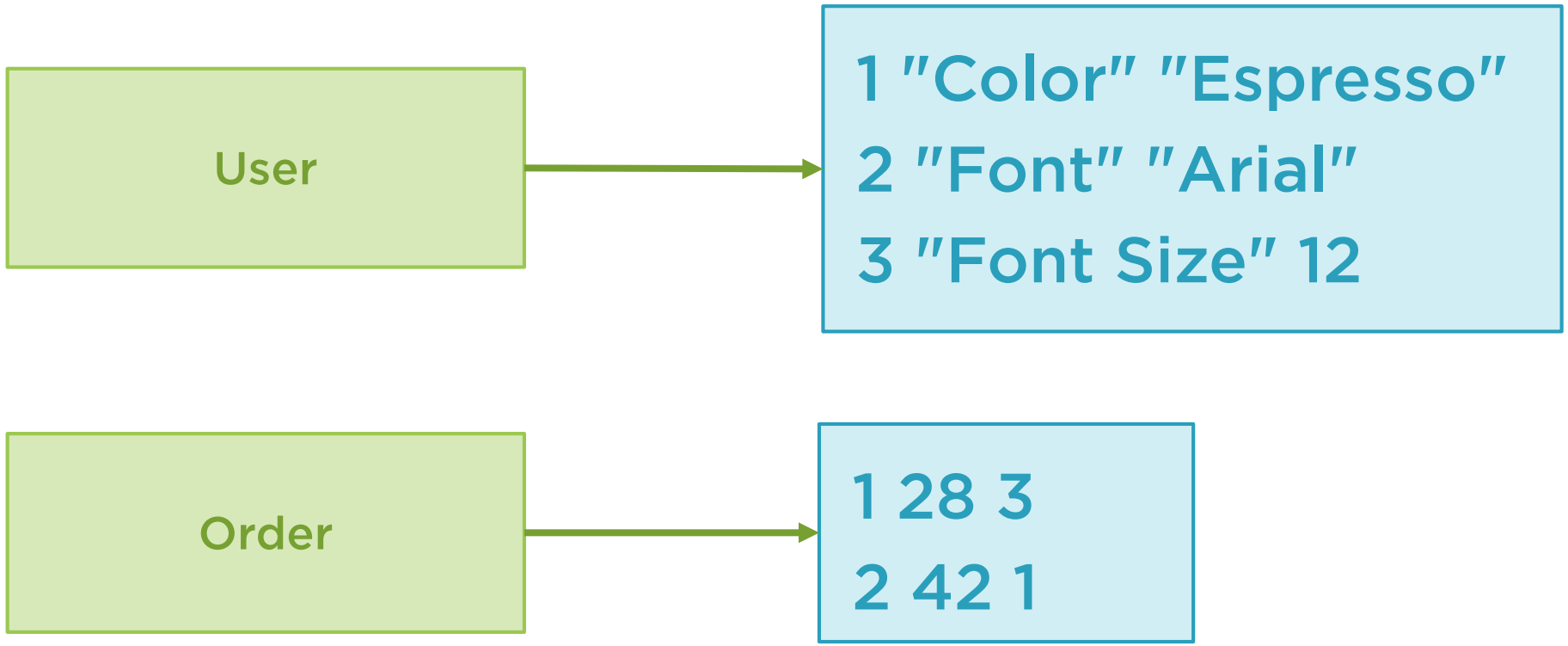
Collection Initializers

```
var colorOptions = new List<string>();
```

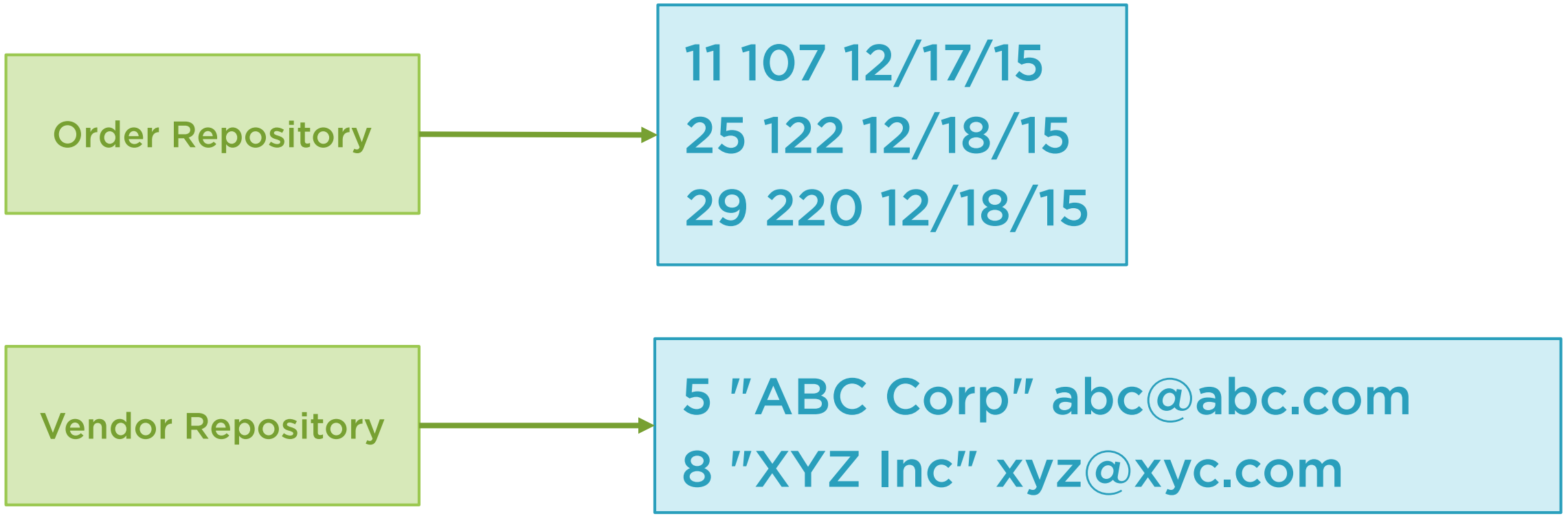
```
colorOptions.Add("Red");
colorOptions.Add("Espresso");
colorOptions.Add("White");
colorOptions.Add("Navy");
```

```
var colorOptions = new List<string>() {"Red","Espresso","White","Navy"};
```

Initializing a List of Objects



Initializing a List of Objects (cont)



Declaring, Initializing, and Populating a List

```
var vendors = new List<Vendor>();
```

```
var vendor = new Vendor() {VendorId=5,CompanyName="ABC Corp",Email="abc@abc.com"};
vendors.Add(vendor);

vendor = new Vendor() {VendorId = 8,CompanyName = "XYZ Inc",Email = "xyz@xyz.com"};
vendors.Add(vendor);
```

```
vendors.Add(new Vendor()
                {VendorId = 5, CompanyName = "ABC Inc", Email = "abc@abc.com"});
vendors.Add(new Vendor()
                {VendorId = 8, CompanyName = "XYZ Inc", Email = "xyz@xyz.com"});
```

Retrieving an Element from a List

```
vendors[1]
```

0	5 "ABC Corp" abc@abc.com
1	8 "XYZ Inc" xyz@xyz.com
2	24 "EFG Ltd" efg@efg.com

Retrieving List Element Best Practices

Do:

Take care when referencing elements by index

Will generate a runtime exception

Avoid:

Retrieving elements by index when you need all elements

Iterate through instead

Iterating Through a Generic List

"Red"
"Espresso"
"White"
"Navy"

5 "ABC Corp" abc@abc.com
8 "XYZ Inc" xyz@xyz.com
24 "EFG Ltd" efg@efg.com

`foreach`

`for`

Iterating a List

`foreach`	`for`
Quick and easy	Complex but flexible
Iterate all elements	Iterate all or a subset of elements
Element is read-only But the element's properties are editable	Element is read/write

Common C# Lists by Namespace

System

- Array

System.Collections (.NET 1)

- ArrayList

System.Collections.Generic (.NET 2+)

- List<T>
- LinkedList<T>
- Queue<T>
- Stack<T>

Selecting an Appropriate List

Array

- Multiple dimensions
- Small performance benefit with large fixed number of elements

ArrayList

- If < .NET 2

List<T>

- Use most often

Selecting an Appropriate List (cont)

LinkedList<T>

- Linked to element before it and after it in sequence
- Insert/remove elements in middle of list

Queue<T>

- Discard element after retrieval
- Access elements in same order as they were added

Stack<T>

- Discard element after retrieval
- Access last added element first

Selecting an Appropriate List (cont)

System.Collections.ObjectModel

- Appropriate for a reusable library
- ReadOnlyCollection
- ObservableCollection

System.Collections.Specialized

- Specialty collections
- StringCollection

System.Collections.Concurrent

- Thread-safe list classes

Selecting a List Best Practices

Do:

Use List<T> unless some other list better meets the requirements

Avoid:

.NET 1 lists in the System.Collections namespace

Such as ArrayList

Frequently Asked Questions

- When is it appropriate to use a generic list?
  - Any time the application needs to manage a list of things.
- What are the key differences between an array and a generic list?
  - An array is fixed length and can have multiple dimensions.
  - A generic list can be any length and provides methods to easily add, insert, or remove elements from the list.

Frequently Asked Questions (cont)

- **What is the difference between `foreach` and `for` when iterating through a list?**
  - `foreach` provides simple syntax for iterating all elements in a list.
  - `for` provides more complex but flexible syntax for iterating all or any subset of elements in a list.
    - Plus the iterated elements are editable.
- **When using `foreach` on a list of objects, is the iterated object editable?**
  - The object instance is not editable.
  - But the object properties are editable.

Summary

Declaring and Populating a Generic List

Using Collection Initializers

Initializing a List of Objects

Retrieving an Element from a Generic List

Iterating Through a Generic List

Types of C# Lists

Generic Dictionaries

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/

Kinds of Collections

Lists

"Red"
"Espresso"
"White"
"Navy"

Dictionaries

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

Generic Dictionary

A strongly typed collection of keys and values

Key:

- Must be unique
- Must not be changed
- Cannot be null

List vs. Dictionary

List	Dictionary
Contains elements	Contains elements defined as key and value pairs
Accessed by a positional index	Accessed by key
Allows duplicate elements	Allows duplicate values but unique keys
Marginally faster iteration	Marginally faster look ups

Generic Dictionary

Dictionary<TKey, TValue>

Dictionary<int, int>

Dictionary<int, string>

Dictionary<string, string>

Dictionary<int, Product>

Dictionary<string, Product>

Overview

- Declaring and Populating a Generic Dictionary
- Using Collection Initializers
- Initializing a Dictionary of Objects
- Retrieving an Element from a Generic Dictionary
- Iterating Through a Generic Dictionary
- Types of C# Dictionaries
- FAQ

Declaring a Generic Dictionary

- **Dictionary of what?**
  - **Value**
  - **Key**

"California"
"Washington"
"New York"

Declaring a Generic Dictionary

```
Dictionary<string, string> states;
```

- Dictionary of what?
  - Value
  - Key
- Dictionary<TKey, TValue>
  - TKey is the type of the key
  - TValue is the type of the value

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

Initializing a Generic Dictionary

```
Dictionary<string, string> states;
states = new Dictionary<string, string>();
```

- Reference type
- new keyword

Declaring and Initializing a Dictionary

```
Dictionary<string, string> states;
states = new Dictionary<string, string>();
```

```
Dictionary<string, string> states =
                            new Dictionary<string, string>();
```

```
var states = new Dictionary<string, string>();
```

Populating a Dictionary

```
states.Add("CA", "California");
```

Populating a Dictionary

```
states.Add("CA", "California");
states.Add("WA", "Washington");
states.Add("NY", "New York");
```

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

Dictionary Best Practices

Do:

Use a generic dictionary to manage a collection by key

Avoid:

Using a dictionary if there is no clear or unique key

Using a dictionary if you don't plan to look elements up by key

Declaring and Populating a Dictionary

```
var states = new Dictionary<string, string>();
```

```
states.Add("CA", "California");
states.Add("WA", "Washington");
states.Add("NY", "New York");
```

Collection Initializers

```
var states = new Dictionary<string, string>();
```

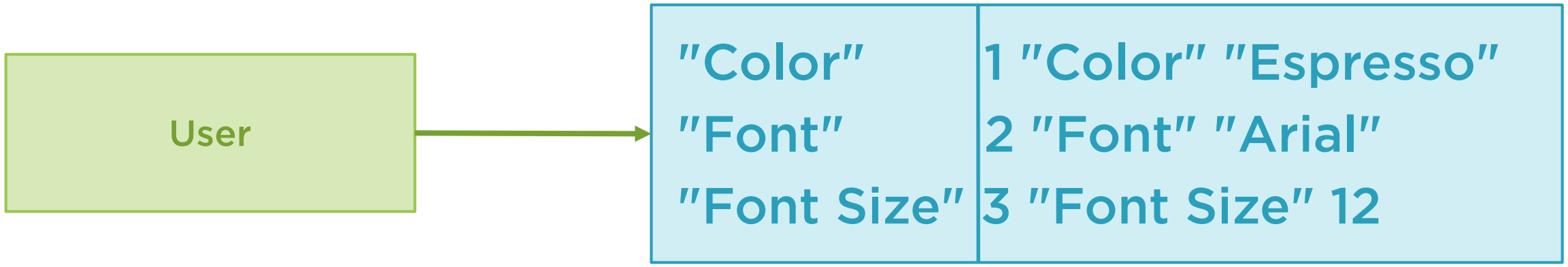
```
states.Add("CA", "California");

states.Add("WA", "Washington");

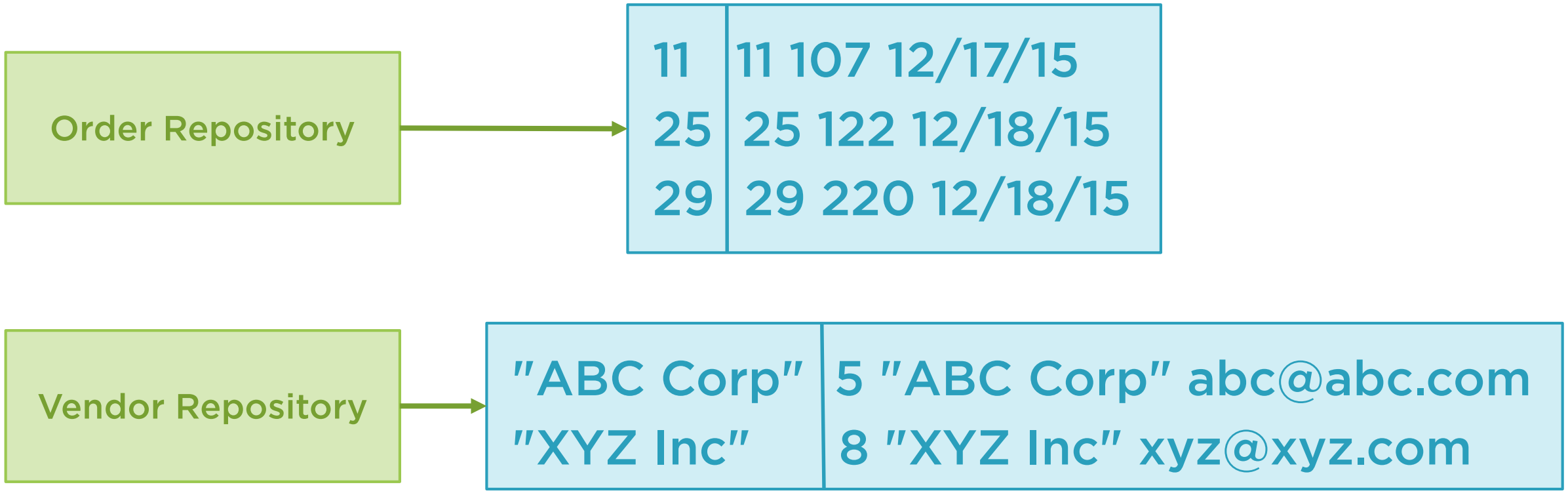
states.Add("NY", "New York");
```

```
var states = new Dictionary<string, string>()
{
    {"CA", "California" },
    {"WA", "Washington"},
    {"NY", "New York" },
};
```

Dictionary of Objects



Dictionary of Objects



Declaring, Initializing, and Populating a Dictionary

```
var vendors = new Dictionary<string,Vendor>();
```

```
var vendor = new Vendor() {VendorId=5,CompanyName="ABC Corp",Email="abc@abc.com"};
vendors.Add(vendor.CompanyName, vendor);

vendor = new Vendor() {VendorId = 8,CompanyName = "XYZ Inc",Email = "xyz@xyz.com"};
vendors.Add(vendor.CompanyName, vendor);
```

Collection Initializers

```
var vendors = new Dictionary<string, Vendor>()
{
    { "ABC Corp", new Vendor()
       { VendorId = 5, CompanyName = "ABC Corp", Email = "abc@abc.com" } },
    { "XYZ Inc", new Vendor()
       { VendorId = 8, CompanyName = "XYZ Inc", Email = "xyz@xyz.com" } }
};
```

Retrieving an Element from a Dictionary

"ABC Corp"	5 "ABC Corp" abc@abc.com
"XYZ Inc"	8 "XYZ Inc" xyz@xyz.com

```
vendors["XYZ Inc"];
```

Retrieving Dictionary Element Best Practices

Do:

Retrieve elements by key

Avoid:

Retrieving elements by key if you are not sure the key is valid

Use ContainsKey or TryGetValue

Retrieving elements by key when you need all elements

Iterate through instead

Iterating Through a Generic Dictionary

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

"ABC Corp"	5 "ABC Corp" abc@abc.com
"XYZ Inc"	8 "XYZ Inc" xyz@xyz.com

- Elements
- Keys
- Values

Iterating Through a Generic Dictionary

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

"ABC Corp"	5 "ABC Corp" abc@abc.com
"XYZ Inc"	8 "XYZ Inc" xyz@xyz.com

`foreach`

Iterating a Dictionary Best Practices

Do:

Use `foreach` to iterate a dictionary

Avoid:

Avoid iterating through the elements

Iterate through the keys or values instead

Common C# Dictionaries by Namespace

System.Collections (.NET 1)

System.Collections.Generic

- Dictionary<TKey,TValue>
- SortedList<TKey,TValue>
- SortedDictionary<TKey,TValue>

Selecting an Appropriate Dictionary

Dictionary<T,V>

- Use most often
- Not sorted

SortedList<T,V>

- Sorted by key
- Faster when populating from sorted data

SortedDictionary <T,V>

- Sorted by key
- Faster when populating from unsorted data

Selecting an Appropriate Dictionary (cont)

System.Collections.ObjectModel	System.Collections.Specialized	System.Collections.Concurrent
• Appropriate for a reusable library • ReadOnlyDictionary • KeyedCollection	• Specialty collections • OrderedDictionary	• Thread-safe dictionary classes

Frequently Asked Questions

- When is it appropriate to use a generic dictionary?
  - Any time the application needs to manage a collection of things by key.
- What are the primary differences between a generic list and a generic dictionary?
  - A generic list contains elements accessible by index.
  - A generic dictionary contains elements with keys, accessible by key.

Frequently Asked Questions (cont)

- **What are the limitations of a dictionary key?**
  - Must be unique within the collection.
  - Must not be changed.
  - Cannot be null.
- **What is the difference between `foreach` and `for` when iterating through a dictionary?**
  - `for` is not useful.
  - `foreach` iterates all elements in a dictionary.
    - Iterate the elements, keys, or values.

Summary

- Declaring and Populating a Generic Dictionary
- Using Collection Initializers
- Initializing a Dictionary of Objects
- Retrieving an Element from a Generic Dictionary
- Iterating Through a Generic Dictionary
- Types of C# Dictionaries

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

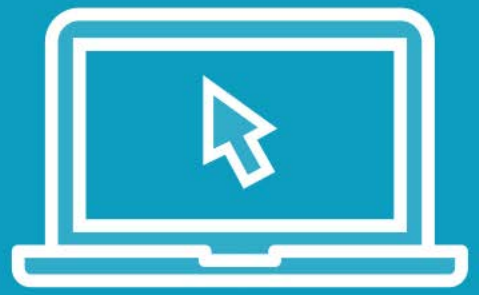
Using an Interface as a Return Type

Returning IEnumerable<T>

Defining an Iterator with `yield`

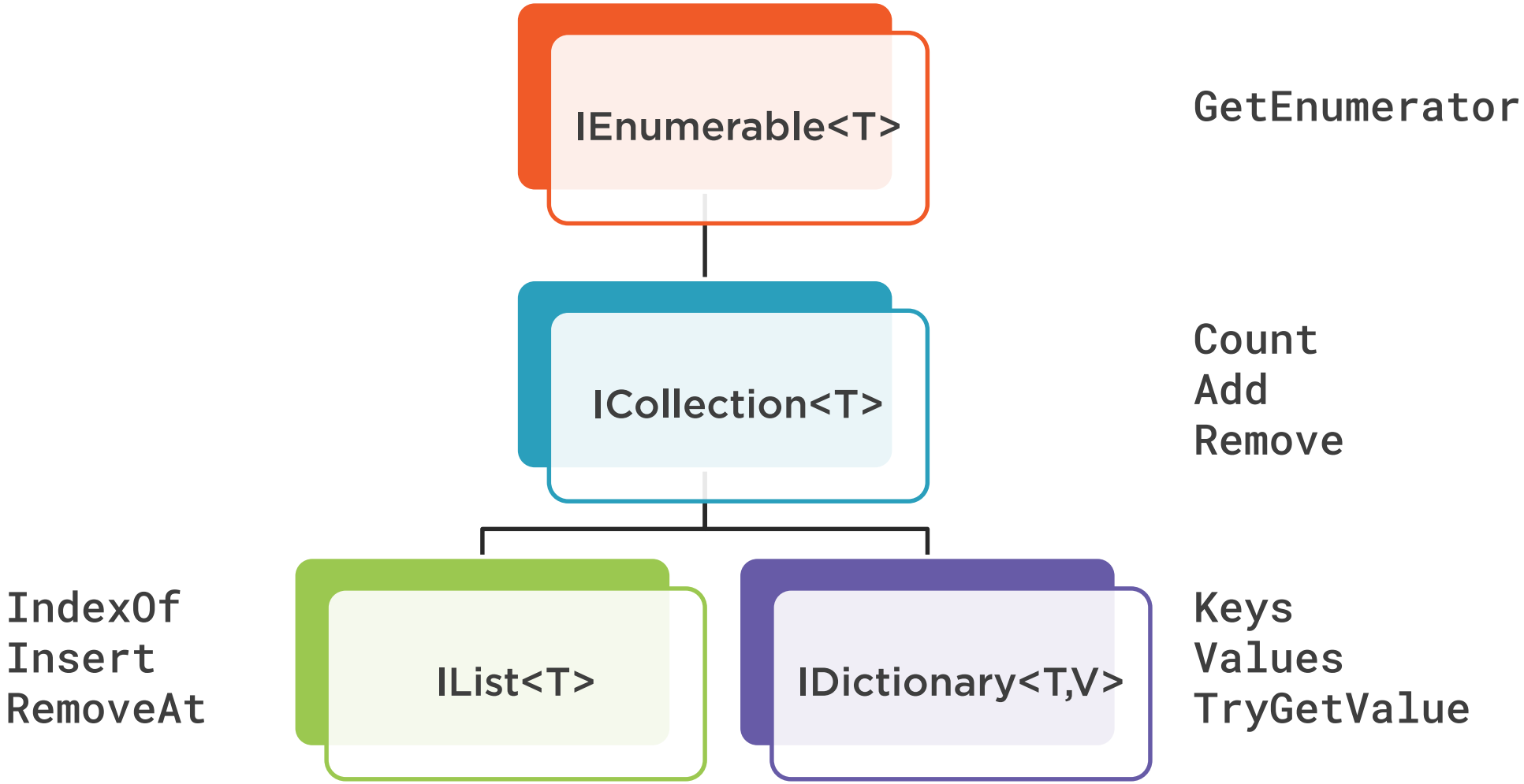
FAQ

Demo

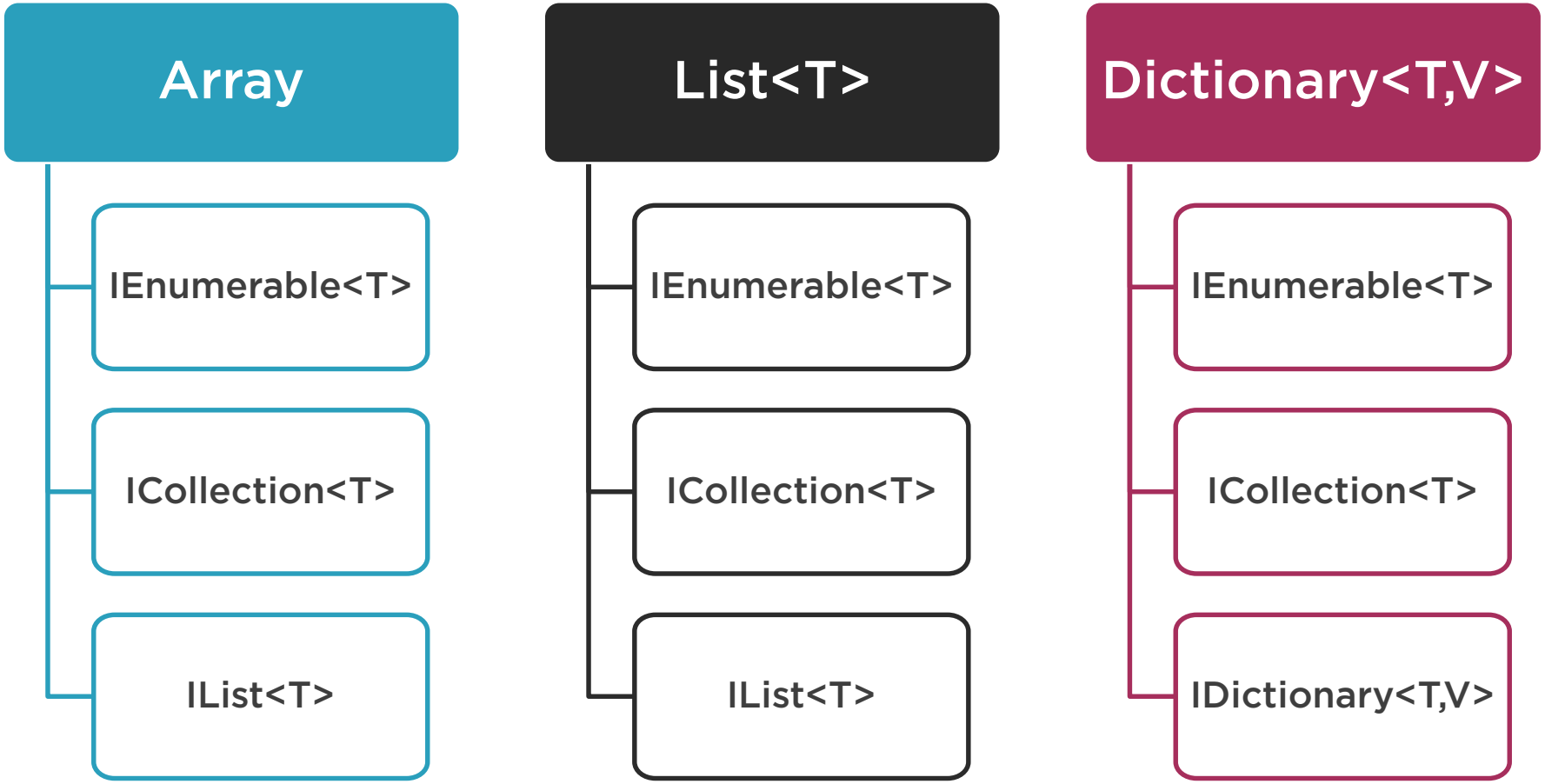


The case for interfaces

Generic Collection Interfaces



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

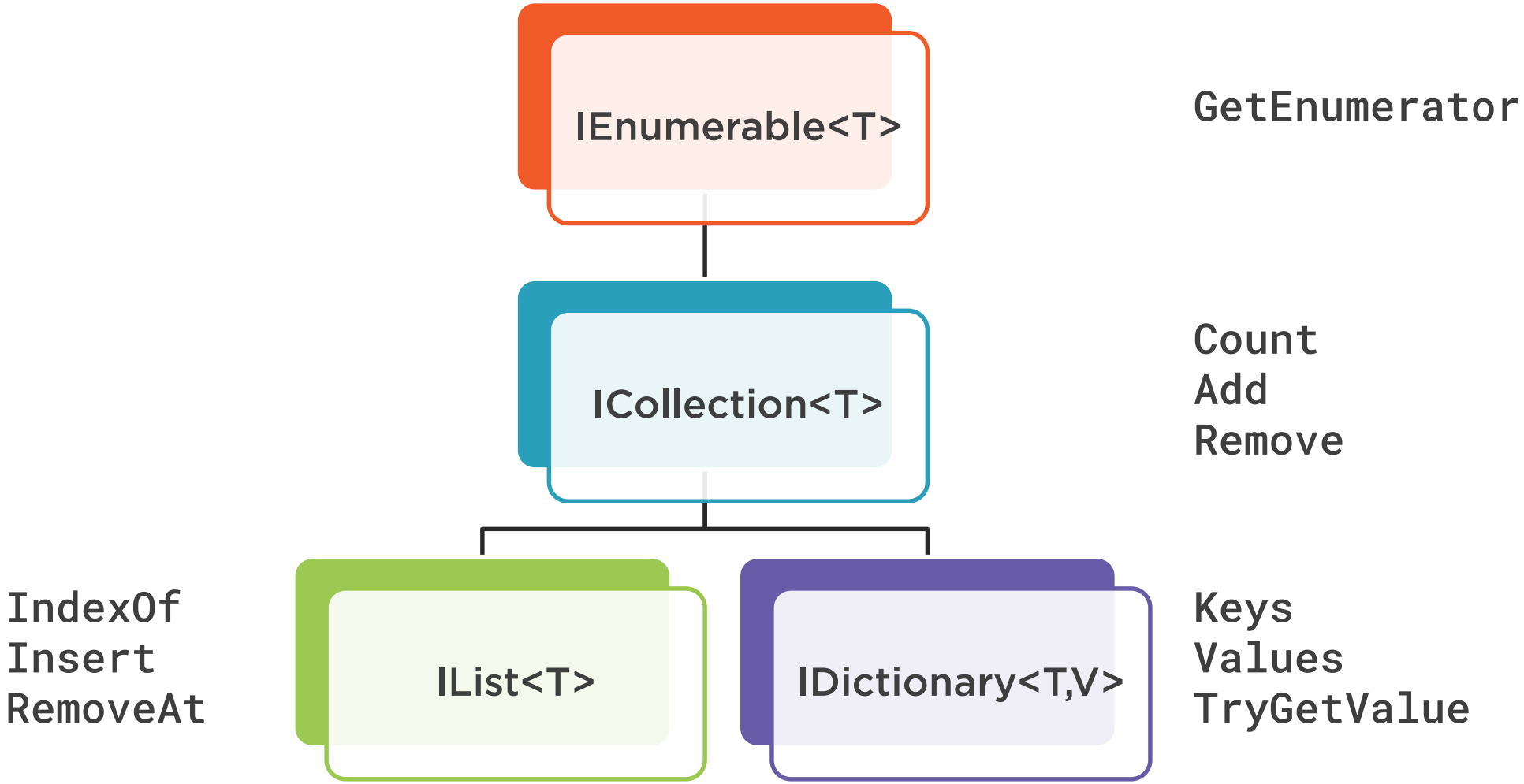
Use a cast operation to cast the result to the desired collection type

Use the most general interface that meets the requirements

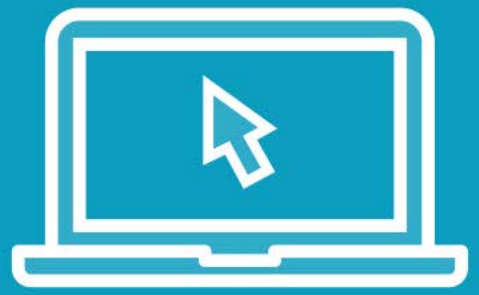
Avoid:

Using any of the interfaces in the System.Collections namespace

IEnumerable<T>



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

LINQ

Deborah Kurata

CONSULTANT | SPEAKER | AUTHOR

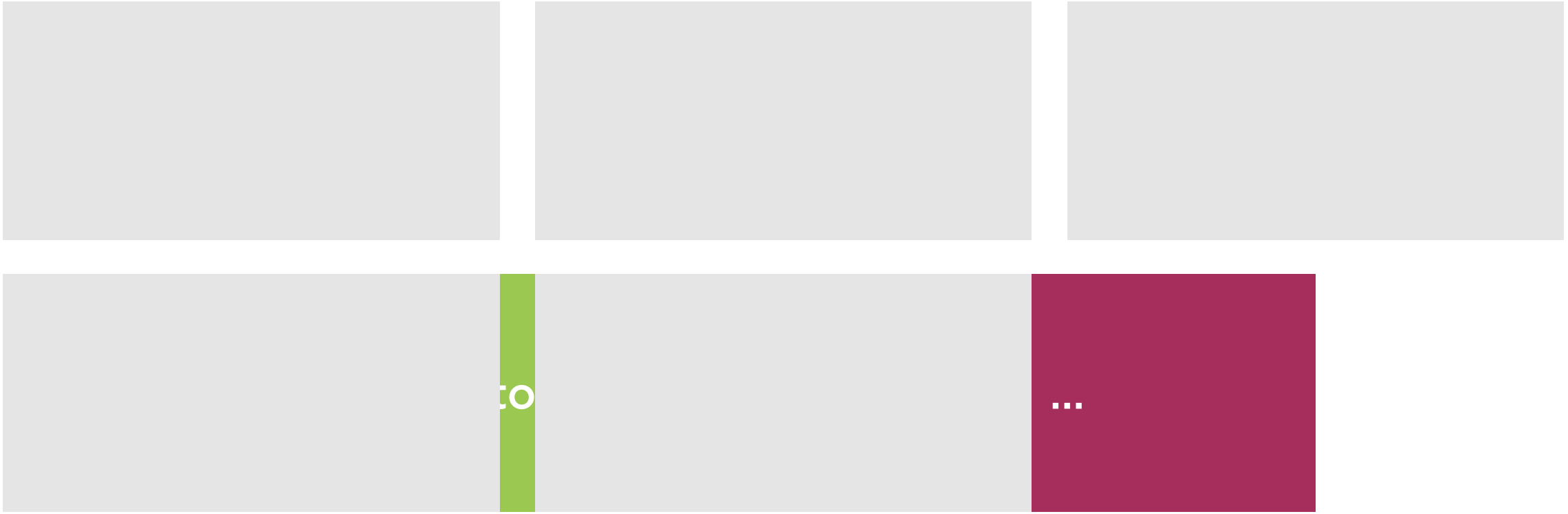
@deborahkurata | blogs.msmvps.com/deborahk/

LINQ

Language INtegrated Query

A way to express queries against a data source directly from a .NET language, such as C#.

Data Sources: LINQ Provider



Query Syntax

```
var vendorQuery = from v in vendors
                  where v.CompanyName.Contains("Toy")
                  orderby v.CompanyName
                  select v;
```

Method Syntax

```
var vendorQuery = vendors
                  .Where(v => v.CompanyName.Contains("Toy"))
                  .OrderBy(v=> v.CompanyName);
```

Extension Method

A method added to an existing type without modifying the original type.

"Object-Oriented Programming Fundamentals in C#"

Delegate

A type that represents a reference to a method with a specific parameter list and return type.

Used to pass methods as arguments to other methods.

```
vendors.Where(Func<Vendor, bool> predicate);

vendors.Where(FilterCompanies);

private bool FilterCompanies(Vendor v)
{
    return v.CompanyName.Contains("Toy");
}
```

Lambda Expression

A method that can be passed as an argument to a method when that argument is expecting a delegate type.

Useful for writing LINQ query expressions.

```
vendors.Where(v => v.CompanyName.Contains("Toy"))
```

Overview

Building a LINQ Query: Query Syntax

Building a LINQ Query: Method Syntax

Using Lambda Expressions

LINQ and Collections

FAQ

Building a LINQ Query: Query Syntax

```
var vendorQuery = from v in vendors
                  where v.CompanyName.Contains("Toy")
                  orderby v.CompanyName
                  select v;
```

Building a LINQ Query: Method Syntax

```
var vendorQuery = vendors.
                          .OrderBy(delegate);

private b                                    ntains("Toy");
```

Array
List<T>
Dictionary<T,V>

IList<T>
ICollection<T>
IEnumerable<T>

Building a LINQ Query: Method Syntax

```
var vendorQuery = vendors.Where(FilterCompanies)
                         .OrderBy(delegate);


private bool FilterCompanies(Vendor v) =>
                        v.CompanyName.Contains("Toy");
```

Building a LINQ Query: Method Syntax

```
var vendorQuery = vendors.Where(FilterCompanies)
                         .OrderBy(OrderCompaniesByName);


private bool FilterCompanies(Vendor v) =>
                        v.CompanyName.Contains("Toy");

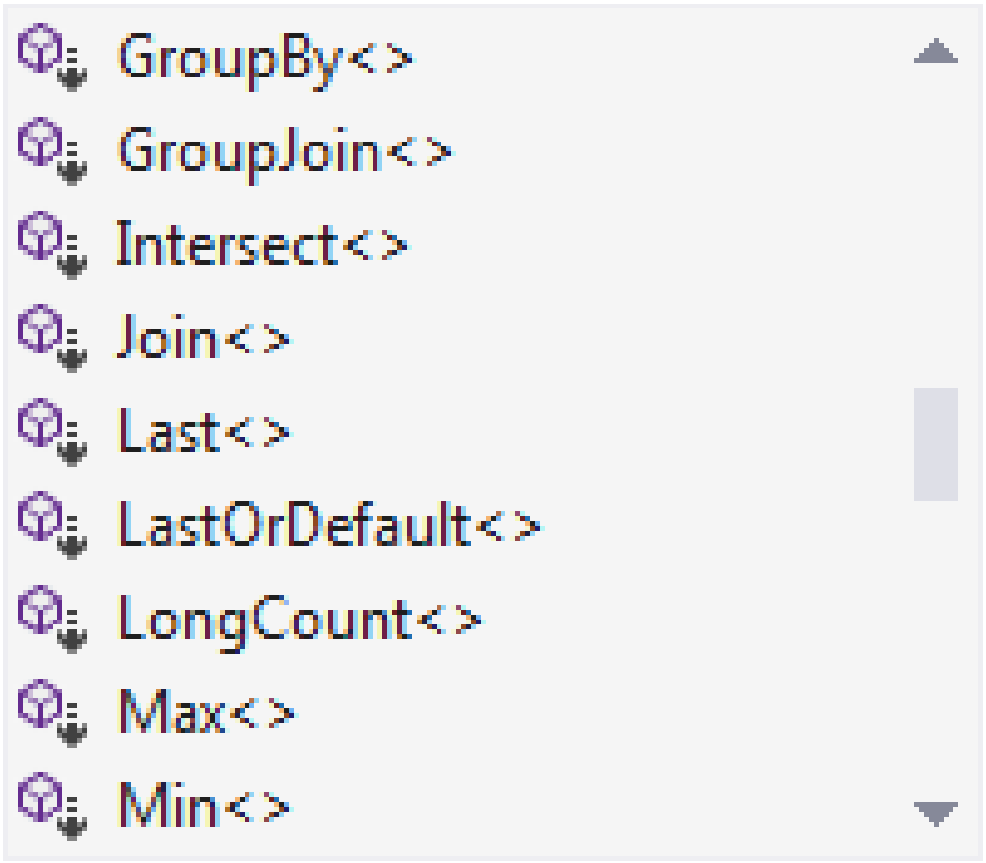
private string OrderCompaniesByName(Vendor v) =>
                        v.CompanyName;
```

Using Lambda Expressions

```
var vendorQuery = vendors
                  .Where(v => v.CompanyName.Contains("Toy"))
                  .OrderBy(v => v.CompanyName);
```

LINQ and Collections

```
vendors.
```



```
vendors.Where(v =>
        v.CompanyName.Contains("Toy")

vendors.OrderBy(v => v.CompanyName)

vendors.Select(v=>v.Email);

vendors.GroupBy(v => v.Category);

vendors.Average(v => v.NoOfProducts);

vendors.First(v => v.VendorId == 22);

vendors.FirstOrDefault(v =>
                  v.VendorId == 22);

vendors.Where(v =>
        v.CompanyName.Contains("Toy"))
  .GroupBy(v => v.Category)
  .Select(g =>
        g.Sum(v => v.NoOfProducts));
```

- Filter based on criteria
- Sort on defined string
- Shape by selecting properties
- Group by a defined property
- Aggregate elements
- Return the first matching element
- Return the first matching element or null
- Combine as needed

LINQ with Collections Best Practices

Do:

Use LINQ!

Consider using the method syntax over the query syntax

Wait to cast the result to a concrete type until all query operations are defined

Avoid:

Iterating over the collection more than is required

Using First or Last methods
Use FirstOrDefault or LastOrDefault instead

Frequently Asked Questions

- What is LINQ?
  - Language INtegrated Query.
  - A way to express queries against a data source from a .NET language such as C#.
- What type of data source works with LINQ?
  - Any data source that has a LINQ provider.
  - LINQ to Objects, LINQ to SQL, LINQ to Entities (Entity Framework)

Frequently Asked Questions (cont)

- What is an extension method?
  - A method added to an existing type without modifying the original type.
  - The LINQ standard query operators are all extension methods.
- What type(s) do the LINQ extension methods extend?
  - IEnumerable and IEnumerable<T>.

Frequently Asked Questions (cont)

- What is a delegate?
  - A data type that represents a reference to a method with a specific parameter list and return type.
  - Used to pass methods as arguments to other methods.
- What is a Lambda expression?
  - A method that can be passed as an argument to a method when that argument expects a delegate type.
  - Are helpful for writing LINQ query expressions.

Summary

Building a LINQ Query: Query Syntax

Building a LINQ Query: Method Syntax

Using Lambda Expressions

LINQ and Collections

For more information on LINQ

"Practical LINQ"

"LINQ Fundamentals"

"Querying the Entity Framework"

Final Words

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/

Takeaways

Collections and Generics

Arrays

Collections and Generics

Arrays

Generics

```
public class OperationResult<T>
{
    public OperationResult() { }

    public OperationResult(T result,
                           string message) : this()
    {
        this.Result = result;
        this.Message = message;
    }

    public T Result { get; set; }
    public string Message { get; set; }
}
```

Collections and Generics

Arrays

Generics

List<T>

Collections and Generics

Arrays

Generics

List<T>

Dictionary<T,V>

Collections and Generics

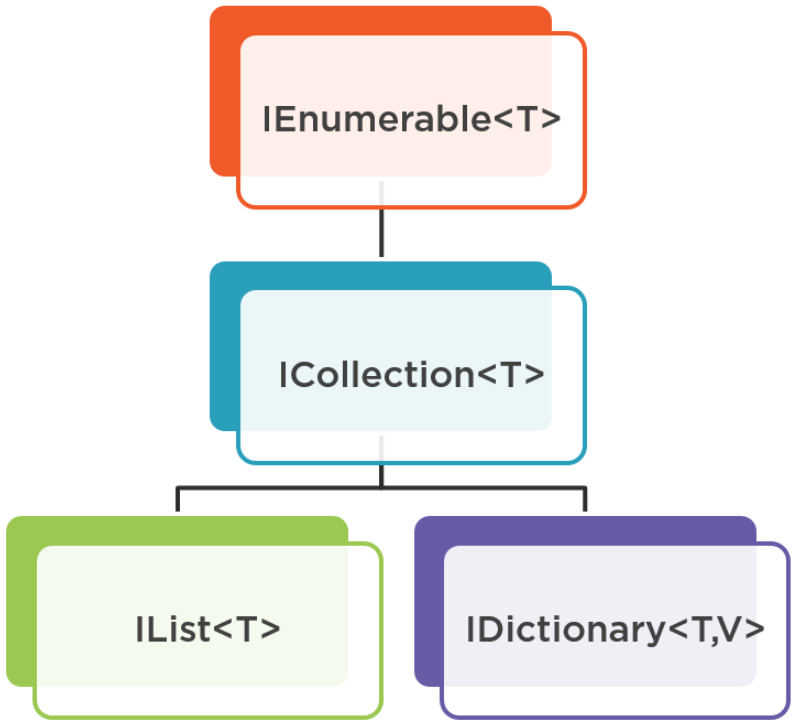
Arrays

Generics

List<T>

Dictionary<T,V>

Interfaces



Collections and Generics

- Arrays
- Generics
- List<T>
- Dictionary<T,V>
- Interfaces
- LINQ

Collections and Generics

- Arrays
- Generics
- List<T>
- Dictionary<T,V>
- Interfaces
- LINQ

'))

For More Information

- Collections
  - C# Collections Fundamentals
  - C# Concurrent Collections
- Generics
  - C# Generics
  - Generics in VB.NET

For More Information (cont)

- **Interfaces**
  - **C# Interfaces**
  - **Object-Oriented Programming Fundamentals in C#**
- **LINQ**
  - **Practical LINQ**
  - **LINQ Fundamentals**
  - **LINQ Data Access**
  - **Querying the Entity Framework**

GitHub

https://github.com/DeborahK/CSharpBP-Collections

C# Best Practices: Collections and Generics

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR

@deborahkurata | blogs.msmvps.com/deborahk/