Microsoft® Official Course

Module 4

Creating Classes and Implementing Type-Safe Collections

Microsoft®

Module Overview

- Creating Classes
- Defining and Implementing Interfaces
- Implementing Type-Safe Collections

- Classes enable you to create your own custom types. They enable you to encapsulate the behaviors and characteristics of any logical entity in a reusable and extensible way.

- Interfaces enable you to define a set of inputs and outputs that classes must implement in order to ensure compatibility with consumers of the classes.

Lesson 1: Creating Classes

- Creating Classes and Members
- Instantiating Classes
- Using Constructors
- Reference Types and Value Types
- Demonstration: Comparing Reference Types and Value Types
- Creating Static Classes and Members
- Testing Classes

Creating Classes and Members

- Use the **class** keyword

```
public class DrinksMachine
{
   // Methods, fields, properties, and events.
}
```

- Specify an access modifier:
  - public: the class is available to code running in any assembly that references the assembly in which the class is contained.
  - internal: the class is available to any code within the same assembly, but not available to code in another assembly. This is the default
  - private: the class is only available to code within the class that contains it. You can only use the private access modifier with nested classes.

- Add methods, fields, properties, and events
  - use fields and properties to define the characteristics
  - use methods to define behavior
  - use events to represent actions that might require your attention

Creating Classes and Members - EXAMPLE

```
public class DrinksMachine
{
    // The following statements define a property with a private field.
    private int _age;
    public int Age
    {
        get
        {
            return _age;
        }
        set
        {
            if (value > 0)
                _age = value;
        }
    }
    // The following statements define properties.
    public string Make;
    public string Model;
    // The following statements define methods.
    public void MakeCappuccino()
    {
        // Method logic goes here.
    }
    public void MakeEspresso()
    {
        // Method logic goes here.
    }
    // The following statement defines an event. The delegate definition is not shown.
    public event OutOfBeansHandler OutOfBeans;
}
```

Instantiating Classes

- A class is just a blueprint for a type.
  - To use the behaviors and characteristics that you define within a class, you need to create instances of the class. An instance of a class is called an object.
    - Try calling methods on references … without new

- To instantiate a class, use the **new** keyword
  - The line below creates a **new object** in memory
  - and it creates an **object reference** that refers to the new object

```
DrinksMachine dm = new DrinksMachine();
```

- To infer the type of the new object, use the **var** keyword

```
var dm = new DrinksMachine();
```

- To call members on the instance, use the dot notation

```
dm.Model = "BeanCrusher 3000";
dm.Age = 2;
dm.MakeEspresso();
```

Using Constructors

- Constructors are a type of **method**:
  - Share the name of the class
  - Called when you instantiate a class
  - often used to specify initial or default values for data members within the new object

- A **default constructor** accepts no arguments
  - **If you do not include any constructor** in your class, the Visual C# compiler will automatically add an empty public default constructor to your compiled class

```
public class DrinksMachine
{
   public void DrinksMachine()
   {
      // This is a default constructor.
   }
}
```

- Classes can include **multiple constructors**

```
var dm1 = new DrinksMachine(2);
var dm2 = new DrinksMachine("Fourth Coffee", "BeanCrusher 3000");
var dm3 = new DrinksMachine(3, "Fourth Coffee", "BeanToaster Turbo");
```

```
public class DrinksMachine
{
   public int Age { get; set; }
   public string Make { get; set; }
   public string Model { get; set; }
   public DrinksMachine(int age)
   {
      this.Age = age;
   }
   public DrinksMachine(string make, string model)
   {
      this.Make = make;
      this.Model = model;
   }
   public DrinksMachine(int age, string make, string model)
   {
      this.Age = age;
      this.Make = make;
      this.Model = model;
   }
```

Reference Types and Value Types

- Value types
  - Contain data directly (e.g.: int, bool, struct)
  - In this case, **First** and **Second** are two distinct items in memory

```
int First = 100;
int Second = First;
```

- Reference types
  - **Point** to an object in memory
  - In this case, **First** and **Second** point to the same item in memory

```
object First = new Object();
object Second = First;
```

- Value types and reference types behave differently.
  - If you copy a value type from one variable to another, you are copying the data that your variable contains and creating a new instance of that data in memory.
  - If you copy an object reference from one variable to another, all you are doing is copying the object reference. You are not creating a second object in memory. Both variables will point to the same object.
    - The **boxing** process is implicit
    - to **unbox** a value type you must explicitly cast

Boxing

```
int i = 100;
object o = i;
```

Unboxing

```
int j;
j = (int)o;
```

Creating Static Classes and Members

- Use the static keyword to create a static class

```
public static class Conversions
{
   // Static members go here.
}
```

- Call members directly on the class name

```
double weightInKilos = 80;
double weightInPounds =
   Conversions.KilosToPounds(weightInKilos);
```

- Static members to non-static classes
  - Non-static classes can include static members.
  - Methods, fields, properties, and events can all be declared static.
  - Static properties are often used to return data that is common to all instances
    - E.g. keep track of how many instances of a class have been created.
  - Static methods are often used to provide utilities that relate to the type in some way
    - E.g. conversion between miles and kilometers (same formula applies regardless of the object/instance …)
  - Regardless of how many instances of your class exist, there is only ever one instance of a static member. You do not need to instantiate the class in order to use static members.

```
public class DrinksMachine
{
   public int Age { get; set; }
   public string Make { get; set; }
   public string Model { get; set; }
   public static int CountDrinksMachines()
   {
      // Add method logic here.
   }
}
```

Testing Classes – unit tests

- In many cases, you will want to test the functionality of your classes in isolation before you integrate them with other classes in your applications.
  - To test functionality in isolation, you create a unit test.
  - A unit test presents the code under test with known inputs, performs an action on the code under test (for example by calling a method), and then verifies that the outputs of the operation are as expected.

- the unit test method is divided into three conceptual phases:
  - Arrange: you create the conditions for the test. You instantiate the class you want to test, and you configure any input values that the test requires.
  - Act: you perform the action that you want to test.
  - Assert: you verify the results of the action. If the results were not as expected, the test fails.

- The Assert.IsTrue method is part of the Microsoft Unit Test Framework that is included in Visual Studio 2012. It throws an exception if the specified condition does not evaluate to true.

Testing Classes – unit tests

```
    private static void Main(string[] args)
    {
        TestGetAge();
    }

    [TestMethod]
    public static void TestGetAge()
    {
        // Arrange.
        DateTime dob = DateTime.Today;
        dob = dob.AddDays(7);
        dob = dob.AddYears(-24);
        Customer testCust = new Customer();
        testCust.DateOfBirth = dob;
        // The customer's 24th birthday is seven days away, so the age in years should be 23.
        int expectedAge = 22;
        // Act.
        int actualAge = testCust.GetAge();
        // Assert.
        // Fail the test if the actual age and the expected age are different.
        Assert.IsTrue(actualAge == expectedAge, "Age not calculated correctly");
    }
}

public class Customer
{
    public DateTime DateOfBirth { get; set; }
    public int GetAge()
    {
        TimeSpan difference = DateTime.Now.Subtract(DateOfBirth);
        int ageInYears = (int)(difference.Days / 365.25);
        // Note: converting a double to an int rounds down to the nearest whole number.
        return ageInYears;
    }
}
```

```
C:\WINDOWS\system32\cmd.exe

Unhandled Exception: Microsoft.VisualStudio.TestTools.UnitTesting.AssertFailedException: Assert.IsTrue failed. Age not c
alculated correctly
   at Microsoft.VisualStudio.TestTools.UnitTesting.Assert.HandleFail(String assertionName, String message, Object[] para
meters)
   at Microsoft.VisualStudio.TestTools.UnitTesting.Assert.IsTrue(Boolean condition, String message)
   at Assignment4Competition1.Program.TestGetAge() in C:\Users\Razvan\source\repos\Test\Test\Program.cs:line 32
   at Assignment4Competition1.Program.Main(String[] args) in C:\Users\Razvan\source\repos\Test\Test\Program.cs:line 14
Press any key to continue . . .
```

```
// The customer's 24th birthday is
int expectedAge = 23;
// Act.
int actualAge = testCust.GetAge();
```

```
C:\WINDOWS\system32\cmd.exe
Press any key to continue . . .
```

https://stackoverflow.com/questions/13602508/where-to-find-microsoft-visualstudio-testtools-unittesting-missing-dll

Lesson 2: Defining and Implementing Interfaces

- Introducing Interfaces
- Defining Interfaces
- Implementing Interfaces
- Implementing Multiple Interfaces
- Implementing the IComparable Interface
- Implementing the IComparer Interface

Introducing Interfaces

- Interfaces define a set of characteristics and behaviors
  - Member signatures only
  - No implementation details
  - Cannot be instantiated
- Interfaces are implemented by classes or structs
  - Implementing class or struct must implement every member
  - Implementation details do not matter to consumers
  - Member signatures must match definitions in interface
- You can think of an interface as a contract.
  - By implementing an interface, a class/struct guarantees that it will provide certain functionality
  - Programming convention: all interface names should begin with an "I".

```
public interface ILoyaltyCardHolder
{
    int TotalPoints { get; }
    int AddPoints(decimal transactionValue);
    void ResetPoints();
}
```

Defining Interfaces

- Use the **interface** keyword

```
public interface IBeverage
{
   // Methods, properties, events, and indexers.
}
```

- Specify an access modifier:
  - public: the interface is available to code running in any assembly
  - internal: the interface is available to any code within the same assembly (default …).

- Add interface members:
  - Methods, properties, events, and indexers
    - Interface members do not include access modifiers. All interface members are public
    - Interfaces cannot include members that relate to the internal functionality of a class, such as fields, constants, operators, and constructors.
  - Signatures only, no implementation details
    - To define a method, you specify the name, the return type, and any parameters:
      - int GetServingTemperature(bool includesMilk);
    - To define a property, you specify the name, the type, and the property accessors:
      - bool IsFairTrade { get; set; }
    - To define an event, you use the **event** keyword, followed by the handler delegate, and the name of the event:
      - event EventHandler OnSoldOut;
    - To define an indexer, you specify the return type and the accessors:
      - string this[int index] { get; set; }

Implementing Interfaces

- Add the name of the interface to the class declaration

```
public class Coffee : IBeverage
```

- Implement all interface members
  - Your class can include additional members that are not defined by the interface

- Interface polymorphism (polymorphism = one key pillar of OOP):
  - represent an instance of a class as an instance of any interface that the class implements
  - if several classes implement the IBeverage interface, such as Coffee, Tea, Juice, and so on you can write code that works with any of these classes as instances of IBeverage …

- Use the interface type and the derived class type interchangeably

```
Coffee coffee1 = new Coffee();
IBeverage coffee2 = new Coffee();//representing an object as an interface type
```

The **coffee2** variable will only expose members defined by the **IBeverage** interface

- You must use an explicit cast to convert from an interface type to a derived class type:
  - IBeverage beverage = coffee1; //implicit cast in here
  - Coffee coffee3 = beverage as Coffee; //explicit cast in here
    // OR
    Coffee coffee4 = (Coffee)beverage; //explicit cast in here

Implementing Multiple Interfaces

- Add the names of each interface to the class declaration

```
public class Coffee : IBeverage, IInventoryItem
```

- Implement every member of every interface
  - Use explicit implementation if two interfaces have a member with the same name

```
// This is an implicit implementation.
public bool IsFairTrade { get; set; }

//These are explicit implementations.
public bool IInventoryItem.IsFairTrade { get; }
public bool IBeverage.IsFairTrade { get; set; }
```

example

- Student class
  - Name
  - Major
  - GPA
- Course class
  - Name
  - Capacity
  - Students
- IPrintable interface …
  - DisplayToConsole()

```
public class CoffeeRatingComparer : IComparer
{
    public int Compare(Object x, Object y)
    {
        Coffee coffee1 = x as Coffee;
        Coffee coffee2 = y as Coffee;
        double rating1 = coffee1.AverageRating;
        double rating2 = coffee2.AverageRating;
        //make use of Double.CompareTo method
        return rating1.CompareTo(rating2);
    }
}
```

- Create a method that works with both: Students and Courses …

Implementing the IComparable Interface

- If you want instances of your class to be sortable in collections, implement the **IComparable** interface

```
public interface IComparable
{
   int CompareTo(Object obj);
}
```

- How does the **ArrayList** instance know how items in the collection should be ordered when it contains objects of type **Coffee**?
  - The **ArrayList.Sort** method calls the **IComparable.CompareTo** method on collection members to sort items in a collection.
  - the **Coffee** class needs to provide the **ArrayList** instance with logic that enables it to compare one coffee with another. For this, the **Coffee** class must implement the **IComparable** interface.

- CompareTo compares the current object instance with another object of the same type (the argument). It should return:
  - **Less than zero** indicates that the current object instance precedes the supplied instance in the sort order.
  - **Zero** indicates that the current object instance occurs at the same position as the supplied instance in the sort order.
  - **More than zero** indicates that the current object instance follows the supplied instance in the sort order.

```
public class Coffee : IComparable
{
    public double AverageRating { get; set; }
    public string Variety { get; set; }
    int IComparable.CompareTo(object obj)
    {
        Coffee coffee2 = obj as Coffee;
        return String.Compare(this.Variety, coffee2.Variety);
    }
}
```

Implementing the IComparer Interface

- To sort collections by custom criteria, implement the **IComparer** interface

```
public interface IComparer
{
   int Compare(Object x, Object y);
}
```

- To use an **IComparer** implementation to sort an **ArrayList**, pass an **IComparer** instance to the **ArrayList.Sort** method

```
ArrayList coffeeList = new ArrayList();
// Add some items to the collection.
coffeeList.Sort(new CoffeeRatingComparer());
```

- To sort the **ArrayList** using a custom comparer, you call the **Sort** method and pass in a **new** instance of your **IComparer** implementation as an argument.

```
public class CoffeeRatingComparer : IComparer
{
    public int Compare(Object x, Object y)
    {
        Coffee coffee1 = x as Coffee;
        Coffee coffee2 = y as Coffee;
        double rating1 = coffee1.AverageRating;
        double rating2 = coffee2.AverageRating;
        //make use of Double.CompareTo method
        return rating1.CompareTo(rating2);
    }
}
```

```
// Create some instances of the Coffee class.
Coffee coffee1 = new Coffee();
coffee1.Rating = 4.5;
Coffee coffee2 = new Coffee();
coffee2.Rating = 8.1;
Coffee coffee3 = new Coffee();
coffee3.Rating = 7.1;
// Add the Coffee instances to an ArrayList.
ArrayList coffeeList = new ArrayList();
coffeeList.Add(coffee1);
coffeeList.Add(coffee2);
coffeeList.Add(coffee3);
// Sort the ArrayList by average rating.
coffeeList.Sort(new CoffeeRatingComparer());
```

Lesson 3: Implementing Type-Safe Collections

- Introducing Generics
- Advantages of Generics
- Constraining Generics
- Using Generic List Collections
- Using Generic Dictionary Collections
- Using Collection Interfaces
- Creating Enumerable Collections
- Demonstration: Adding Data Validation and Type-Safety to the Application Lab

Introducing Generics

- Generics enable you to create and use strongly typed collections that are type safe, do not require you to cast items, and do not require you to box and unbox value types.

- Generic classes work by including a type parameter, T, in the class or interface declaration.
  - You do not need to specify the type of T until you instantiate the class.

- To create a generic class/interface, you need to:
  - Add the type parameter T in angle brackets after the class name.
  - Use the type parameter T in place of type names in your class members.

```
public class CustomList<T>
{
   public T this[int index] { get; set; }
   public void Add(T item) { ... }
   public void Remove(T item) { ... }
}
```

- Specify the type argument when you instantiate the class

```
CustomList<Coffee> coffees =    new CustomList<Coffee>();
```

Advantages of Generics over non-generic types

- Type safety
  - In an ArrayList one could add objects of type Coffee and (by mistake) Tea without compiling error.
  - Then, at runtime, when you cast the objects from the list into Coffee an invalid cast runtime exception will occur.
  - Instead of **ArraList** use **List<T>** Example …

- No casting
  - Casting is a computationally expensive process. When you add items to an **ArrayList**, your items are implicitly cast to the **System.Object** type.
  - When you retrieve items from an **ArrayList**, you must explicitly cast them back to their original type.
  - Using generics to add and retrieve items without casting improves the performance of your application.

- No boxing and unboxing
  - If you want to store value types in an ArrayList, the items must be boxed when they are added to the collection and unboxed when they are retrieved.

```
int number1 = 1;
var arrayList1 = new ArrayList();
// This statement boxes the Int32 value as a System.Object.
arrayList1.Add(number1);
// This statement unboxes the Int32 value.
int number2 = (int)arrayList1[0];
var genericList1 = new List<Int32>();
//This statement adds an Int32 value without boxing.
genericList1.Add(number1);
//This statement retrieves the Int32 value without unboxing.
int number3 = genericList1[0];
```

  - Boxing and unboxing incurs a large computational cost and can significantly slow your applications, especially when you iterate over large collections.
  - By contrast, you can add value types to generic lists without boxing and unboxing the value.

Constraining Generics

Constraint	Description
where T : <name of interface>	The type argument must be, or implement, the specified interface.
where T : <name of base class>	The type argument must be, or derive from, the specified class.
where T : U	The type argument must be, or derive from, the supplied type argument U.
where T : new()	The type argument must have a public default constructor.
where T : struct	The type argument must be a value type.
where T : class	The type argument must be a reference type.

Apply Multiple Type Constraints

```
public class CustomList<T> where T : IBeverage, IComparable<T>, new()
{
}
```

Using Generic List Collections

Generic list classes store collections of objects of type **T**:

- **List<T>** is a general purpose generic list
  - Add an item.
  - Remove an item.
  - Insert an item at a specified index.
  - Sort the items in the collection
    - using the default comparer or a specified comparer.
  - Reorder all or part of the collection.

```
string s1 = "Latte";
string s2 = "Espresso";
string s3 = "Americano";
string s4 = "Cappuccino";
string s5 = "Mocha";
// Add the items to a strongly-typed collection.
var coffeeBeverages = new List<String>();
coffeeBeverages.Add(s1);
coffeeBeverages.Add(s2);
coffeeBeverages.Add(s3);
coffeeBeverages.Add(s4);
coffeeBeverages.Add(s5);
// Sort the items using the default comparer.
// For objects of type String, the default comparer sorts the items alphabetically.
coffeeBeverages.Sort();
// Write the collection to a console window.
foreach(String coffeeBeverage in coffeeBeverages)
{
   Console.WriteLine(coffeeBeverage);
}
Console.ReadLine ("Press Enter to continue");
```

- **LinkedList<T>** is a generic list in which each item is linked to the previous item and the next item in the collection
- **Stack<T>** is a last in, first out collection
- **Queue<T>** is a first in, first out collection

- You should use these classes instead of non-generic collection classes whenever possible.

Using Generic Dictionary Collections

- Generic dictionary classes store key-value pairs
  - Both the key and the value are strongly typed

- **Dictionary<TKey, TValue>**
  - Is a general purpose, generic dictionary class.
  - You can add duplicate values to the collection, but the keys must be unique.
  - The class will throw an **ArgumentException** if you attempt to add a key that already exists in the dictionary.

- **SortedList<TKey, TValue>** and **SortedDictionary<TKey, TValue>**
  - collections are sorted by key
  - **SortedList** generic class uses less memory than the **SortedDictionary** generic class.
  - **SortedDictionary** class is faster and more efficient at inserting and removing unsorted data

Using Generic Dictionary Collections

```
// Create a new dictionary of strings with string keys.
var coffeeCodes = new Dictionary<String, String>();
// Add some entries to the dictionary.
coffeeCodes.Add("CAL", "Café Au Lait");
coffeeCodes.Add("CSM", "Cinammon Spice Mocha");
coffeeCodes.Add("ER", "Espresso Romano");
coffeeCodes.Add("RM", "Raspberry Mocha");
coffeeCodes.Add("IC", "Iced Coffee");
// This statement would result in an ArgumentException because the key already exists.
// coffeeCodes.Add("IC", "Instant Coffee");
// To retrieve the value associated with a key, you can use the indexer.
// This will throw a KeyNotFoundException if the key does not exist.
Console.WriteLine("The value associated with the key \"CAL\" is {0}",
coffeeCodes["CAL"]);
// Alternatively, you can use the TryGetValue method.
// This returns true if the key exists and false if the key does not exist.
string csmValue = "";
if (coffeeCodes.TryGetValue("CSM", out csmValue))
{
    Console.WriteLine("The value associated with the key \"CSM\" is {0}", csmValue);
}
else
{
    Console.WriteLine("The key \"CSM\" was not found");
}
// You can also use the indexer to change the value associated with a key.
coffeeCodes["IC"] = "Instant Coffee";
```

Using Collection Interfaces

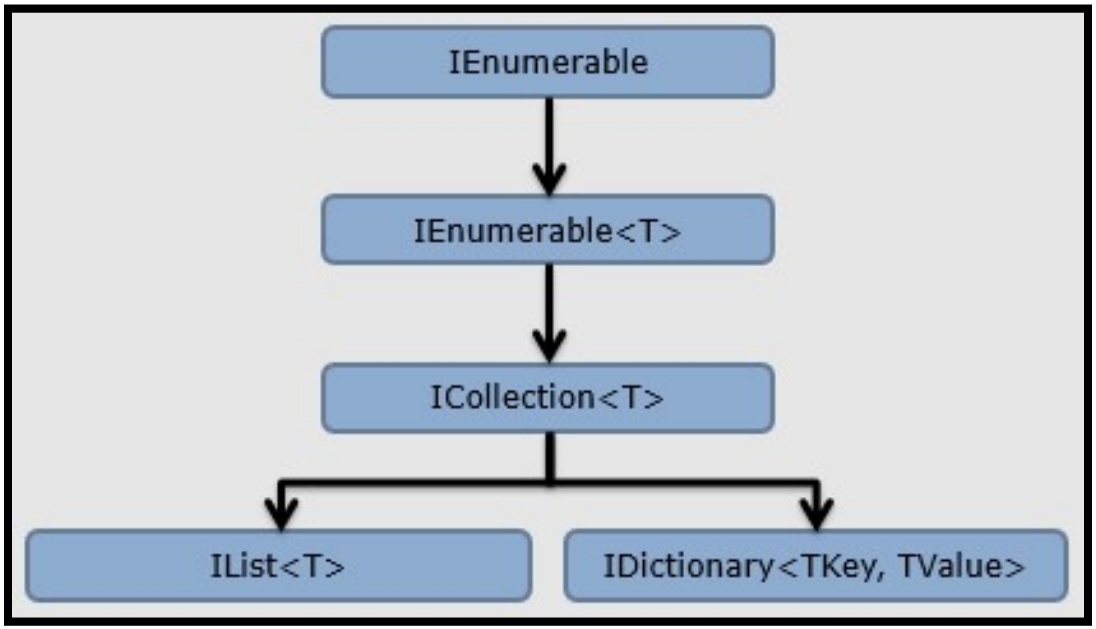
- **custom collection** classes: You might want to store data in a tree structure or create a circular linked list …

- If you want to be able to use a foreach loop to enumerate over the items in your custom generic collection, you must implement the **IEnumerable<T>** interface
  - defines a single method **GetEnumerator()** that returns an object of type **IEnumerator<T>**.

- The **IEnumerable<T>** interface inherits from the **IEnumerable** interface, which also defines a single method named **GetEnumerator()**.
  - If you implement **IEnumerable<T>**, you also need to implement **IEnumerable**.

- **ICollection<T>** interface defines the basic functionality that is common to all generic collections.
  - inherited methods …
  - Add, Clear
  - Contains
  - CopyTo
  - Remove, Count
  - IsReadOnly

- **IList<T>** interface defines the core functionality for generic list classes
  - inherited methods …
  - Insert
  - RemoveAt
  - IndexOf
  - indexers … list[0]



- **IDictionary<TKey, TValue>** interface defines the core functionality for generic dictionary classes … (Add, ContainsKey, GetEnumerator, Remove, Keys, Values← Icollection<T>)

Creating Enumerable Collections

- Implement **IEnumerable<T>** to support enumeration (**foreach**)
  - **IEnumerator<T>** interface defines the functionality that all enumerators must implement. Methods: **MoveNext**, **Reset**; Properties: **Current**.
    - An **enumerator** is essentially a pointer to the items in the collection.
    - The starting point for the pointer is before the first item.
    - When you call the **MoveNext** method, the pointer advances to the next element in the collection. The **MoveNext** method returns **true** if the enumerator was able to advance one position, or **false** if it has reached the end of the collection.
    - At any point during the enumeration, the **Current** property returns the item to which the enumerator is currently pointing
  - **IEnumerable<T>** interface exposes a method, **GetEnumerator**, which must return an **IEnumerator<T>** instance.
    - The **GetEnumerator** method returns the default enumerator for your collection class. This is the enumerator that a **foreach** loop will use, unless you specify an alternative. However, you can create additional methods to expose alternative enumerators.
    - Implement the **GetEnumerator** method by either:
      - Creating an **IEnumerator<T>** implementation
      - Using an iterator: use the **yield return** statement to implement an iterator

Implementing an Enumerator by Using an Iterator

- You can provide an **enumerator** by creating a custom class that implements the **IEnumerator<T>** interface.
  - As seen on previous slides …
- However, if your custom collection class uses an **underlying enumerable type** to store data, you can **use an iterator** to implement the IEnumerable<T> interface without actually providing an IEnumerator<T> implementation
- Within the foreach loop, a **yield return** statement is used to return each item in the collection.

```
using System;
using System.Collections;
using System.Collections.Generic;
class BasicCollection<T> : IEnumerable<T>
{
    private List<T> data = new List<T>();
    public void FillList(params T [] items)
    {
        foreach (var datum in items)
          data.Add(datum);
    }
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {
        foreach (var datum in data)
        {
            yield return datum;
        }
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }
}
```

Module Review and Takeaways

- Review Question(s)
- **Question**: Which of the following types is a reference type?
  - ( )Option 1: Boolean
  - ( )Option 2: Byte
  - ( )Option 3: Decimal
  - ( )Option 4: Int32
  - ( )Option 5: Object
- **Question**: Which of the following types of member CANNOT be included in an interface?
  - ( )Option 1: Events
  - ( )Option 2: Fields
  - ( )Option 3: Indexers
  - ( )Option 4: Methods
  - ( )Option 5: Properties
- **Question** You want to create a custom generic class. The class will consist of a linear collection of values, and will enable developers to queue items from either end of the collection. Which of the following should your class declaration resemble?
  - ( )Option 1: public class DoubleEndedQueue<T> : IEnumerable<T>
  - ( )Option 2: public class DoubleEndedQueue<T> : ICollection<T>
  - ( )Option 3: public class DoubleEndedQueue<T> : IList<T>
  - ( )Option 4: public class DoubleEndedQueue<T> : IList<T>, IEnumerable<T>
  - ( )Option 5: public class DoubleEndedQueue<T> : IDictionary<TKey,TValue>