Module Overview

Module Overview

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

Overview

Classes enable you to create your own custom, self-contained, and reusable types. 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. In this module, you will learn how to use interfaces and classes to define and create your own custom, reusable types. You will also learn how to create and use enumerable, type-safe collections of any type.

Objectives

After completing this module, you will be able to:

- Create and instantiate classes.
- Create and instantiate interfaces.
- Use generics to create type-safe collections.

Lesson 1: Creating 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

Lesson Overview

In Visual C#, you can define your own custom types by creating *classes*. As a programming construct, the class is central to object-oriented programming in Visual C#. It enables you to encapsulate the behaviors and characteristics of any logical entity in a reusable and extensible way.

In this lesson, you will learn how to create, use, and test classes in your own applications.

OBJECTIVES

After completing this lesson, you will be able to:

- Create classes.
- Create objects by instantiating classes.
- Use constructors to set values or to run logic when classes are instantiated.
- Explain the difference between reference types and value types.
- Create static classes and members.
- Describe the high-level process for testing class functionality.

Creating Classes and Members

Creating Classes and Members

- Use the **class** keyword

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

- Specify an access modifier:
  - public
  - internal
  - private
- Add methods, fields, properties, and events

In Visual C#, a *class* is a programming construct that you can use to define your own custom types. When you create a class, you create a new type, which is effectively creating a blueprint for the instance. The class defines the behaviors and characteristics, or class members, which exists in all instances of the class. You represent these behaviors and characteristics by defining methods, fields, properties, and events within your class.

Declaring a Class

For example, suppose you create a class named **DrinksMachine**.

You use the **class** keyword to declare a class, as shown in the following example:

Declaring a Class

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

The **class** keyword is preceded by an *access modifier, such as* **public** in the above example, which specifies where you can use the type. You can use the following access modifiers in your class declarations:

Access modifier	Description
public	The type is available to code running in any assembly that references the assembly in which the class is contained.
internal	The type is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier.
private	The type is only available to code within the class that contains it. You can only use the private access modifier with nested classes.

Adding Members to a Class

You would use fields and properties to define the characteristics of a drinks machine, such as the make, model, age, and service interval of the machine. You would create methods to represent the things that a drinks machine can do, such as make an espresso or make a cappuccino. Finally, you would define events to represent actions that might require your attention, such as replacing coffee beans when the machine has run out of coffee beans.

Within your class, you can add methods, fields, properties, and events to define the behaviors and characteristics of your type, as shown in the following example:

Defining Class Members

```
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

Instantiating Classes

- To instantiate a class, use the **new** keyword

```
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();
```

A class is just the definition of a type. To use the behaviors and characteristics that you define within a class, you need to create *instances* of the class.

To create a new instance of a class, you use the **new** keyword, as shown in the following example:

Instantiating a Class

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

When you instantiate a class in this way, you are actually doing two things:

- You are creating a new *object* in memory based on the **DrinksMachine** type.
- You are creating an *object reference* named **dm** that refers to the new **DrinksMachine** object.

When you create your object reference, instead of explicitly specifying the **DrinksMachine** type, you can allow the compiler to deduce the type of the object at compile time. This is known as type inference. To use type inference, you create your object reference by using the **var** keyword, as shown in the following example:

Instantiating a Class by Using Type Inference

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

In this case, the compiler does not know in advance the type of the **dm** variable. When the **dm** variable is initialized as a reference to a **DrinksMachine** object, the compiler deduces that the type of **dm** is **DrinksMachine**. Using type inference in this way causes no change in how your application runs; it is simply a shortcut for you to avoid typing the class name twice. In some circumstances, type inference can make your code easier to read, while in other circumstances it may make your code more confusing. As a general rule, consider using type inference when the type of variable is absolutely clear.

After you have instantiated your object, you can use any of the members—methods, fields, properties, and events—that you defined within the class, as shown in the following example:

Using Object Members

```
var dm = new DrinksMachine();
dm.Make = "Fourth Coffee";
dm.Model = "Beancrusher 3000";
dm.Age = 2;
dm.MakeEspresso();
```

This approach to calling members on an instance variable is known as *dot notation*. You type the variable name, followed by a period, followed by the member name. The IntelliSense feature in Visual Studio will prompt you with member names when you type a period after a variable.

Using Constructors

- Constructors are a type of method:
  - Share the name of the class
  - Called when you instantiate a class
- A default constructor accepts no arguments

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

- Classes can include multiple constructors
- Use constructors to initialize member variables

In the previous topics, you might have noticed that the syntax for instantiating a class—for example, **new DrinksMachine()**—looks similar to the syntax for calling a method. This is because when you instantiate a class, you are actually calling a special method called a *constructor*. A constructor is a method in the class that has the same name as the class.

Constructors are often used to specify initial or default values for data members within the new object, as shown by the following example:

Adding a Constructor

```
public class DrinksMachine
{
public int Age { get; set; }
public DrinksMachine()
{
Age = 0;
}
}
```

A constructor that takes no parameters is known as the *default constructor*. This constructor is called whenever someone instantiates your class without providing any arguments. If you do not include a constructor in your class, the Visual C# compiler will automatically add an empty public default constructor to your compiled class.

In many cases, it is useful for consumers of your class to be able to specify initial values for data members when the class is instantiated. For example, when someone creates a new instance of **DrinksMachine**, it might be useful if they can specify the make and model of the machine at the same time. Your class can include multiple constructors with

different signatures that enable consumers to provide different combinations of information when they instantiate your class.
The following example shows how to add multiple constructors to a class:

Adding Multiple Constructors

```
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;
}
}
```

Consumers can use any of the constructors to create instances of your class, depending on the information that is available to them at the time. For example:

Calling Constructors

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

Reference Types and Value Types

Reference Types and Value Types

- Value types
  - Contain data directly

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

  - In this case, **First** and **Second** are two distinct items in memory
- Reference types
  - Point to an object in memory

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

  - In this case, **First** and **Second** point to the same item in memory

Now that you know how to create and instantiate classes, you will learn about the differences between classes and structs.
First, however, let's see how Visual C# works behind the scenes.

The Common Language Runtime (CLR)

The .Net framework consists of several components, built one upon the other. The first is the CLR. This is the engine that envelops and runs all .Net applications. The CLR is responsible for - among other things: the execution of .Net code, Memory management and garbage collection for the application, exception handling, etc.
Above the CLR is the Base Class Library (BCL), that includes all the built-in types in the .Net Framework.
Above that are the specific implementation of the different languages .Net supports.

Additional Reading: For more information on the CLR, see: https://aka.ms/moc-20483c-m4-pg3.

The .Net Memory Model

Now let's address how variables are actually saved in the computer memory. .Net uses two main methods of saving its working memory.

The Stack

The stack is used to save local value variables in the current scope. It functions just like the stack collection, with each new variable being assigned right next to the last one used. When the code exits its current scope, all the associated data in the stack is cleared, and the stack reverts to its previous scope. A scope in .Net is generally defined as the currently running code block surrounded by curly brackets. The most common use of that is the method. When a method is entered, the stack will save all local value variables created in that method, as well as any value parameters passed to the method. When the code exits the method, all of this data is cleared. However, if another method is called from the current method, the new data will be saved right after the current data. This allows the stack to easily load the appropriate data when a scope is exited.

The stack has a fixed size during the lifetime of the application. Exceeding it will result in a **StackOverflowException.**

The Heap

The heap is used to save large reference variables. Memory is allocated dynamically from the heap as needed. When a new large object is created, the CLR looks for the next continuous free memory space large enough to contain the object and allocates that memory to it.

Memory is not automatically cleared from the heap when the scope is exited like it was with the stack. Rather, the garbage collector is responsible to clear the heap. When it decides so, the garbage collector will initiate a scan of the objects in the heap. It continues to delete any object it determines is no longer in use. Lastly the garbage collector defragments the memory left in use to eliminate small gaps between existing objects. This is to avoid waste since objects on the heap are only allocated to continuous free memory spaces.

The heap doesn't have a fixed size and can grow during the lifetime of the application. However, it does have an upper limit to its size. If the application reaches that upper limit, it throws an **OutOfMemoryException** exception**.**

Define Reference and Value Types

Now that we know where and how the CLR stores our application's variables, let see what it saves where and how to interact with them

Reference Types

These are assumed to be rather large objects and are therefore referred to by their address only (hence the name). When you create a reference type object, you do allocate all the memory it requires, however, what you receive back is not the object itself, but only its address in the memory. When you "copy" a reference type, as shown below:

```
Var x = new MyClass();
Var y = x;
```

Only the reference is copied. Any change made to data within the instance will show when either x or y is queried, since they are, in fact, the same instance.

```
x.MyProp = 42;
Console.WriteLine(y.MyProp); // Print 42
y.MyProp += 100;
Console.WriteLine(x.MyProp); // Print 142
```

This also means any change made to a referenced type passed as a parameter to a method will be present when the method exits. Reference types are only saved on the heap.

In Visual C#, classes and delegates, are always reference types. To create a new reference type yourself, you only need to define a new class. This applies to built-in types as well, you can see if a certain type is a reference type by looking up its definition.

Value Types

These are assumed to be smaller objects and are not referenced indirectly. When a value type variable is created, that variable will keep the variable's value itself. When you copy a value type to another it really is copied, resulting in two different instances with the same value. A change to the first will not affect the other.

```
Var x = 1;
Var y = x;
x = 42
Console.WriteLine(x); // Print 42
Console.WriteLine(y); // Print 1
y += 100;
Console.WriteLine(x); // Print 42
Console.WriteLine(y); // Print 101
```

When a value type is assigned as a method parameter, its value is again copied, and the method will use the copied value. Meaning that any change to the parameter's value will not be seen when the method returns.

Value types are sometimes saved on the heap, and sometimes on the stack, depending on their usage. Generally, only local method members and parameters are saved on the stack. While other usages (like class fields and properties) are saved on the heap as a part of their parent object.
In Visual C#, only **structs** are value types. To create a new value type yourself, you only need to define a new **struct**. This applies to built-in types as well, you can see if a certain type is a value type by looking up its definition.
Note: Most of the built-in types in Visual C#, such as int, bool, byte, and char, are value types. For more information about built-in types, see the Built-In Types Table (C# Reference) page at http://go.microsoft.com/fwlink/?LinkID=267800.

Boxing and Unboxing

In some scenarios you may need to convert value types to reference types, and vice versa. For example, some collection classes will only accept reference types. This is less likely to be an issue with the advent of generic collections. However, you still need to be aware of the concept, because a fundamental concept of Visual C# is that you can treat any type as an object.
The process of converting a value type to a reference type is called boxing. To box a variable, you assign it to an object reference:

Boxing

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

The boxing process is implicit. When you assign an object reference to a value type, the Visual C# compiler automatically creates an object to wrap the value and stores it in memory. If you copy the object reference, the copy will point to the same object wrapper in memory.
The process of converting a reference type to a value type is called unboxing. Unlike the boxing process, to unbox a value type you must explicitly cast the variable back to its original type:

Unboxing

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

Note: Note that boxing and unboxing is a slow and expensive process. It's usually better keep the type specified if possible. Using generics is a good solution to the problem. Learn more about generics in Lesson 3 - Implementing Type-Safe Collections.

Demonstration: Comparing Reference Types and Value Types

Demonstration: Comparing Reference Types and Value Types

In this demonstration, you will learn how to:

- Create a value type to store an integer value
- Create a reference type to store an integer value
- Observe the differences in behavior when you copy the value type and the reference type

In this demonstration, you will create a value type to store an integer value and a reference type to store an integer value. You will create a copy of each type, and then observe what happens when you change the value of the copy in each case.

Demonstration Steps

You will find the steps in the "Demonstration: Comparing Reference Types and Value Types" section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_DEMO.md.

Creating Static Classes and Members

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);
```

- Add static members to non-static classes

In some cases, you may want to create a class purely to encapsulate some useful functionality, rather than to represent an instance of anything. For example, suppose you wanted to create a set of methods that convert imperial weights and measures to metric weights and measures, and vice versa. It would not make sense if you had to instantiate a class in order to use these methods, because you do not need to store or retrieve any instance-specific data. In fact, the concept of an instance is meaningless in this case.

In scenarios like this, you can create a static class. A static class is a class that cannot be instantiated. To create a static class, you use the **static** keyword. Any members within the class must also use the **static** keyword, as shown in the following example:

Static Classes

```
public static class Conversions
{
public static double PoundsToKilos(double pounds)
{
// Convert argument from pounds to kilograms
double kilos = pounds * 0.4536;
return kilos;
}
public static double KilosToPounds(double kilos)
{
// Convert argument from kilograms to pounds
double pounds = kilos * 2.205;
return pounds;
}
}
```

To call a method on a static class, you call the method on the class name itself instead of on an instance name, as shown by the following example:

Calling Methods on a Static Class

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

Static Members

Non-static classes can include static members. This is useful when some behaviors and characteristics relate to the instance (instance members), while some behaviors and characteristics relate to the type itself (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, or to 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, such as comparison functions.

To declare a static member you use the **static** keyword before the return type of the member, as shown by the following example:

Static Members in Non-static Classes

```
public class DrinksMachine
{
public int Age { get; set; }
public string Make { get; set; }
public string Model { get; set; }
```

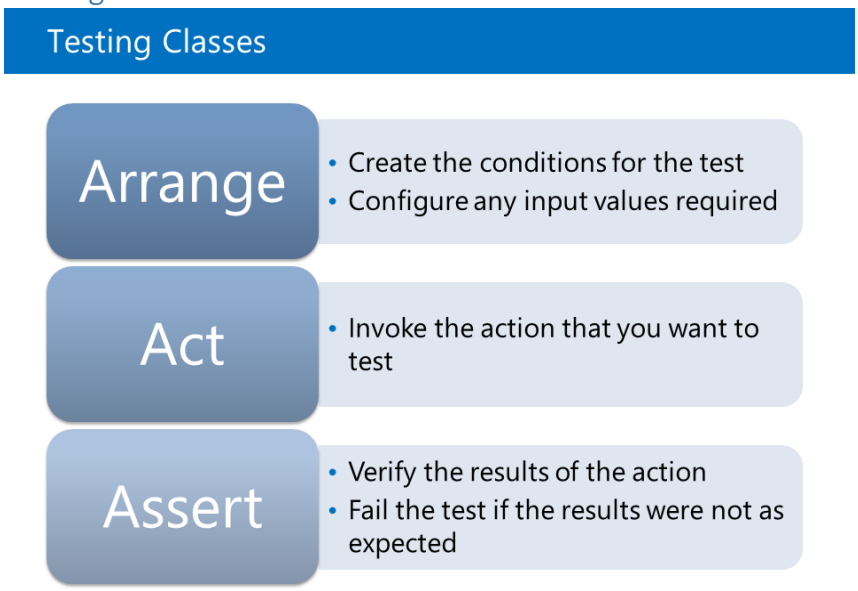
```
public static int CountDrinksMachines()
{
// Add method logic here.
}
}
```

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. You access static members through the class name rather than the instance name, as shown by the following example:

Access Static Members

```
int drinksMachineCount = DrinksMachine.CountDrinksMachines();
```

Testing Classes



Classes often represent self-contained units of functionality. 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. In this way, the unit test represents a contract that your code must fulfill. However, when you change the implementation of a class or method, the unit test ensures that your code always returns particular outputs in response to particular inputs.

For example, consider the case where you create a simple class to represent a customer. To help you target your marketing efforts, the Customer class includes a **GetAge** method that returns the current age of the customer in years:

Class Under Test

```
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;
}
}
```

In this case, you might want to create a unit test that ensures the **GetAge** method behaves as expected. As such, your test method needs to instantiate the **Customer** class, specify a date of birth, and then verify that the **GetAge** method returns the correct age in years. Depending on the unit test framework you use, your test method might look something like the following:

Example Test Method

```
[TestMethod]
public void TestGetAge()
{
// Arrange.
DateTime dob = DateTime.Today;
dob.AddDays(7);
```

```
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 = 23;
// 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");
}
```

Notice that the unit test method is divided into three conceptual phases:

- **Arrange**. In this phase, 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**. In this phase, you perform the action that you want to test.
- **Assert**. In this phase, 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. This particular method throws an exception if the specified condition does not evaluate to **true**. However, the concepts described here are common to all unit testing frameworks.

Lesson 2: Defining and Implementing Interfaces

Lesson 2: Defining and Implementing Interfaces

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

Lesson Overview

An *interface* is a little bit like a class without an implementation. It specifies a set of characteristics and behaviors by defining signatures for methods, properties, events, and indexers, without specifying how any of these members are implemented. When a class implements an interface, the class provides an implementation for each member of the interface. By implementing the interface, the class is thereby guaranteeing that it will provide the functionality specified by the interface.

In this lesson, you will learn how to define and implement interfaces.

OBJECTIVES

After completing this lesson, you will be able to:

- Explain the purpose of interfaces in Visual C#.
- Create interfaces.
- Create classes that implement a single interface.
- Create classes that implement multiple interfaces.
- Implement the **IComparable** interface.
- Implement the **IComparer** interface.

Introducing Interfaces

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
- By implementing an interface, a class or struct guarantees that it will provide certain functionality

In Visual C#, an interface specifies a set of characteristics and behaviors; it does this by defining methods, properties, events, and indexers. The interface itself does not specify how these members are implemented. Instead, classes can *implement* the interface and provide their own implementations of the interface members. You can think of an interface as a contract. By implementing a particular interface, a class guarantees to consumers that it will provide specific functionality through specific members, even though the actual implementation is not part of the contract.
For example, suppose that you want to develop a loyalty card scheme for Fourth Coffee. You might start by creating an interface named **ILoyaltyCardHolder** that defines:

- A read-only integer property named **TotalPoints**.
- A method named **AddPoints** that accepts a decimal argument.
- A method named **ResetPoints**.

The following example shows an interface that defines one read-only property and two methods:

Defining an Interface

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

Note: Programming convention dictates that all interface names should begin with an "I".
Notice that the methods in the interface do not include method bodies. Similarly, the properties in the interface indicate which accessors to include but do not provide any implementation details. The interface simply states that any implementing class must include and provide an implementation for the three members. The creator of the implementing class can choose how the methods are implemented. For example, any implementation of the **AddPoints** method will accept a decimal argument (the cash value of the customer transaction) and return an integer (the number of points added). The class developer could implement this method in a variety of ways. For example, an implementation of the **AddPoints** method could:

- Calculate the number of points to add by multiplying the transaction value by a fixed amount.
- Get the number of points to add by calling a service.
- Calculate the number of points to add by using additional factors, such as the location of the loyalty cardholder.

The following example shows a class that implements the **ILoyaltyCardHolder** interface:

Implementing an Interface

```
public class Customer : ILoyaltyCardHolder
{
private int totalPoints;
public int TotalPoints
{
get { return totalPoints; }
}
public int AddPoints(decimal transactionValue)
{
int points = Decimal.ToInt32(transactionValue);
totalPoints += points;
```

```
}
public void ResetPoints()
{
totalPoints = 0;
}
// Other members of the Customer class.
}
```

The details of the implementation do not matter to calling classes. By implementing the **ILoyaltyCardHolder** interface, the implementing class is indicating to consumers that it will take care of the **AddPoints** operation. One of the key advantages of interfaces is that they enable you to modularize your code. You can change the way in which your class implements the interface at any point, without having to update any consumer classes that rely on an interface implementation.

Defining Interfaces

Defining Interfaces

- Use the **interface** keyword

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

- Specify an access modifier:
  - public
  - internal
- Add interface members:
  - Methods, properties, events, and indexers
  - Signatures only, no implementation details

The syntax for defining an interface is similar to the syntax for defining a class.
You use the **interface** keyword to declare an interface, as shown by the following example:

Declaring an Interface

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

Similar to a class declaration, an interface declaration can include an *access modifier*. You can use the following access modifiers in your interface declarations:

Access modifier	Description
public	The interface is available to code running in any assembly.
internal	The interface is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier.

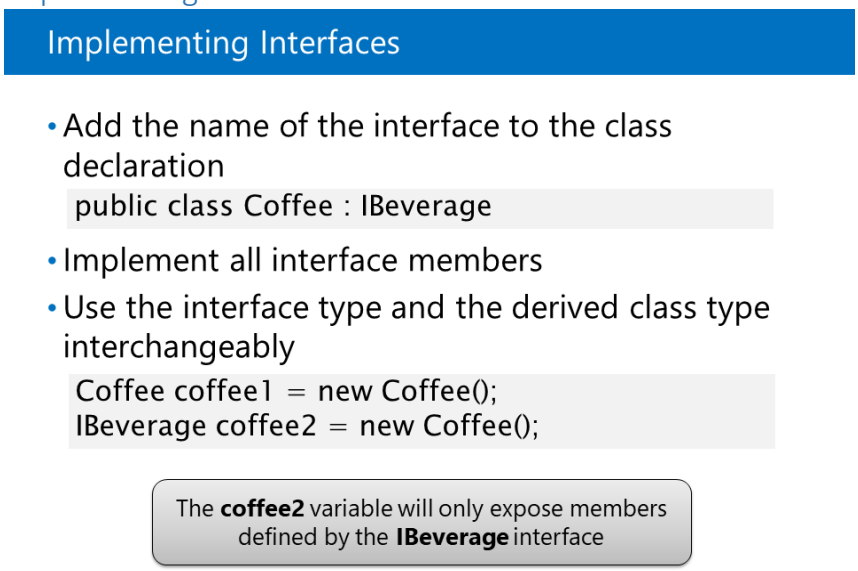
Adding Interface Members

An interface defines the signature of members but does not include any implementation details. Interfaces can include methods, properties, events, and indexers:

- To define a method, you specify the name of the method, the return type, and any parameters:
  `int GetServingTemperature(bool includesMilk);`
- To define a property, you specify the name of the property, the type of the property, and the property accessors:
  `bool IsFairTrade { get; set; }`
- To define an event, you use the **event** keyword, followed by the event handler delegate, followed by 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; }`

Interface members do not include access modifiers. The purpose of the interface is to define the members that an implementing class should expose to consumers, so that 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.

Implementing Interfaces



To create a class that implements an interface, you add a colon followed by the name of the interface to your class declaration.

The following example shows how to create a class that implements the **IBeverage** interface:

Declaring a Class that Implements an Interface

```
public class Coffee : IBeverage
{
}
```

Within your class, you must provide an implementation for *every* member of the interface. Your class can include additional members that are not defined by the interface. In fact, most classes will include additional members, because generally the class *extends* the interface. However, you cannot omit any interface members from the implementing class. The way you implement the interface members does not matter, as long as your implementations have the same *signatures* (that is, the same names, types, return types, and parameters) as the member definitions in the interface.

The following example shows a trivial interface, together with a class that implements the interface:

Implementing an Interface

```
public interface IBeverage
{
int GetServingTemperature(bool includesMilk);
bool IsFairTrade { get; set; }
}
public class Coffee : IBeverage
{
private int servingTempWithoutMilk { get; set; }
private int servingTempWithMilk { get; set; }
public int GetServingTemperature(bool includesMilk)
{
if(includesMilk)
{
return servingTempWithMilk;
}
else
{
return servingTempWithoutMilk;
}
}
public bool IsFairTrade { get; set; }
// Other non-interface members go here.
}
```

Interface Polymorphism

As it relates to interfaces, polymorphism states that you can represent an instance of a class as an instance of any interface that the class implements. Interface polymorphism can help to increase the flexibility and modularity of your

code. Suppose you have several classes that 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**, without knowing any details of the implementing class. For example, you can build a collection of **IBeverage** instances without needing to know the details of every class that implements **IBeverage**.

For example, If the **Coffee** class implements the **IBeverage** interface, you can represent a new **Coffee** object as an instance of **Coffee** or an instance of **IBeverage:**

Representing an Object as an Interface Type

```
Coffee coffee1 = new Coffee();
IBeverage coffee2 = new Coffee();
```

You can use an implicit cast to convert to an interface type, because you know that the class must include all the interface members.

Casting to an Interface Type

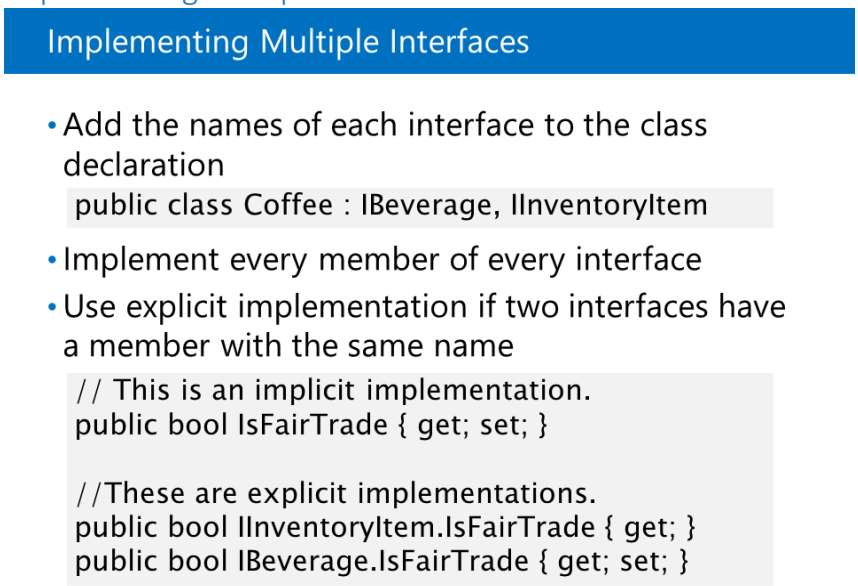
```
IBeverage beverage = coffee1;
```

You must use an explicit cast to convert from an interface type to a derived class type, as the class may include members that are not defined in the interface.

Casting an Interface Type to a Derived Class Type

```
Coffee coffee3 = beverage as Coffee;
// OR
Coffee coffee4 = (Coffee)beverage;
```

Implementing Multiple Interfaces



In many cases, you will want to create classes that implement more than one interface. For example, you might want to:

- Implement the **IDisposable** interface to enable the .NET runtime to dispose of your class correctly.
- Implement the **IComparable** interface to enable collection classes to sort instances of your class.
- Implement your own custom interface to define the functionality of your class.

To implement multiple interfaces, you add a comma-separated list of the interfaces that you want to implement to your class declaration. Your class must implement every member of every interface you add to your class declaration. The following example shows how to create a class that implements multiple interfaces:

Declaring a Class that Implements Multiple Interfaces

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

Implicit and Explicit Implementation

When you create a class that implements an interface, you can choose whether to implement the interface implicitly or explicitly. To implement an interface implicitly, you implement each interface member with a signature that matches the member definition in the interface. To implement an interface explicitly, you fully qualify each member name so that it is clear that the member belongs to a particular interface.

The following example shows an explicit implementation of the **IBeverage** interface:

Implementing an Interface Explicitly

```
public class Coffee : IBeverage
{
```

```
private int servingTempWithoutMilk { get; set; }
private int servingTempWithMilk { get; set; }
public int IBeverage.GetServingTemperature(bool includesMilk)
{
if(includesMilk)
{
return servingTempWithMilk;
}
else
{
return servingTempWithoutMilk;
}
}
public bool IBeverage.IsFairTrade { get; set; }
// Other non-interface members.
}
```

Some developers prefer explicit interface implementation because doing so can make the code easier to understand. The only scenario in which you must use explicit interface implementation is if you are implementing two interfaces that share a member name. For example, if you implement interfaces named **IBeverage** and **IInventoryItem**, and both interfaces declare a Boolean property named **IsAvailable**, you would need to implement at least one of the **IsAvailable** members explicitly. In this scenario, the compiler would be unable to resolve the **IsAvailable** reference without an explicit implementation.

There is a difference in the usage of the class, however. When implementing an interface implicitly the members can be used as normal public members of the class and can be referenced without any special designation. When an interface is implemented explicitly, though, the members are made only available by casting the instance to the corresponding interface. Otherwise, they'll remain hidden, and the code calling them will not compile.

The following example shows the usage of an explicit implementation of the **IBeverage** interface. Assume the Coffee class is the same as the previous example.

Consuming an Explicit Interface implementation

```
Coffee myCoffee = new Coffee();
// This will not compile
myCoffee.GetServingTemperature(true)
IBeverage myBeverage = (IBeverage)myCoffee;
// This will work as expected
myBeverage.GetServingTemperature(true)
```

Implementing the IComparable Interface

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);
}
```

- The **ArrayList.Sort** method calls the **IComparable.CompareTo** method on collection members to sort items in a collection

The .NET Framework includes various collection classes that enable you to sort the contents of the collection. These classes, such as the **ArrayList** class, include a method named **Sort**. When you call this method on an **ArrayList** instance, the collection orders its contents.

How does the **ArrayList** instance know how items in the collection should be ordered? In the case of simple types, such as integers, this appears fairly straightforward. Intuitively, three follows two and two follows one. However, suppose you create a collection of **Coffee** objects. How would the **ArrayList** instance determine whether one coffee is greater or lesser than another coffee? The answer is that the **Coffee** class needs to provide the **ArrayList** instance

with logic that enables it to compare one coffee with another. To do this, the **Coffee** class must implement the **IComparable** interface.

The following example shows the **IComparable** interface:

The IComparable Interface

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

As you can see, the **IComparable** interface declares a single method named **CompareTo**. Implementations of this method must:

- Compare the current object instance with another object of the same type (the argument).
- Return an integer value that indicates whether the current object instance should be placed before, in the same position, or after the passed-in object instance.

The integer values returned by the **CompareTo** method are interpreted as follows:

- 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.

The following example illustrates what happens if you use the **CompareTo** method to compare two integers:

CompareTo Example

```
int number1 = 5;
int number2 = 100;
int result = number1.CompareTo(number2);
// The value of result is -1, indicating that number1 should precede number2 in the sort order.
```

Note: All the built-in value types in the .NET Framework implement the IComparable interface. For more information about the IComparable interface, seethe IComparable Interface page at https://aka.ms/moc-20483c-m4-pg1.

When you call the **Sort** method on an **ArrayList** instance, the **Sort** method calls the **CompareTo** method of the collection members to determine the correct order for the collection.

When you implement the **IComparable** interface in your own classes, you determine the criteria by which objects should be compared. For example, you might decide that coffees should be sorted alphabetically by variety.

The following example shows how to implement the IComparable interface:

Implementing the IComparable Interface

```
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);
}
}
```

In this example, because the values we want to compare are strings, we can use the **String.Compare** method. This method returns -1 if the current string precedes the supplied string in an alphabetical sort order, 0 if the strings are identical, and 1 if the current string follows the supplied string in an alphabetical sort order.

Implementing the IComparer Interface

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());
```

When you call the **Sort** method on an **ArrayList** instance, the **ArrayList** sorts the collection based on the **IComparable** interface implementation in the underlying type. For example, if you sort an **ArrayList** collection of integers, the sort criteria is defined by the **IComparable** interface implementation in the **Int32** type. The creator of the **ArrayList** instance has no control over the criteria that are used to sort the collection.

In some cases, developers may want to sort instances of your class using alternative sort criteria. For example, suppose you want to sort a collection of **Coffee** instances by the value of the **AverageRating** property rather than the **Variety** property. To sort an **ArrayList** instance by using custom sort criteria, you need to do two things:

1. Create a class that implements the **IComparer** interface to provide your custom sort functionality.
2. Call the **Sort** method on the **ArrayList** instance, and pass in an instance of your **IComparer** implementation as a parameter.

The following example shows the **IComparer** interface:

The IComparer Interface

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

As you can see, the **IComparer** interface declares a single method named **Compare**. Implementations of this method must:

- Compare two objects of the same type.
- Return an integer value that indicates whether the current object instance should be placed before, in the same position, or after the passed-in object instance.

The following example shows how to implement the **IComparer** interface:

Implementing the IComparer Interface

```
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;
return rating1.CompareTo(rating2);
}
}
```

In the above example, because the values we want to compare are doubles, we can make use of the **Double.CompareTo** method. This returns -1 if the current double is less than the supplied double, 0 if the current double is equal to the supplied double, and 1 if the current double is greater than the supplied double. It is always better to make use of a built-in comparison function, if one exists, rather than creating your own.

The following example shows how to use a custom **IComparer** implementation:

Using an IComparer Implementation

```
// 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());
```

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.

Lesson 3: Implementing Type-Safe Collections

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

Lesson Overview

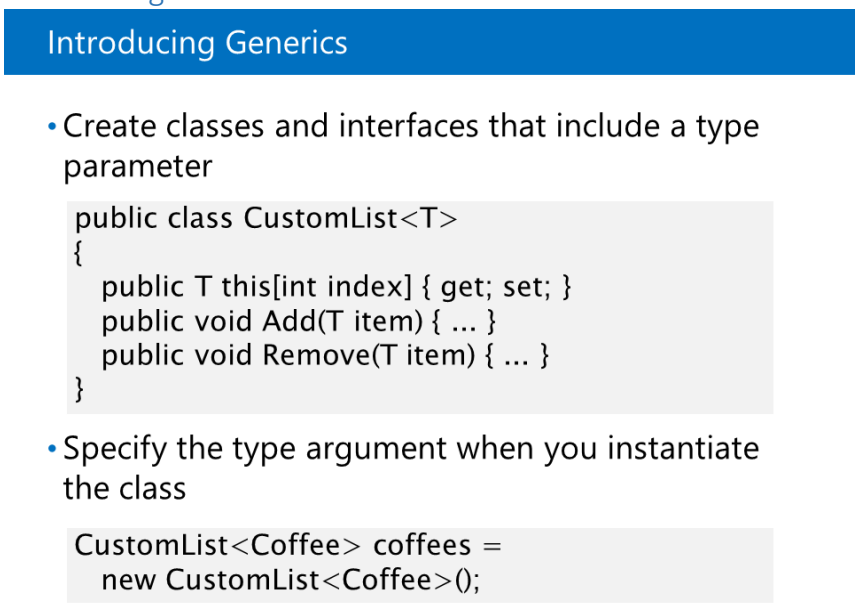
Almost every application that you create will use collection classes in one form or another. In most cases, collections contain a set of objects of the same type. When you interact with a collection, you often rely on the collection to provide objects of a specific type. Historically, this created various challenges. You had to create exception handling logic in case a collection contained items of the wrong type. You also had to box value types in order to add them to collection classes, and unbox them on retrieval. Visual C# removes many of these challenges by using *generics*
In this lesson, you will learn how to create and use generic classes to create strongly typed collections of any type.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe generics.
- Identify the advantages of generic classes over non-generic classes.
- Apply constraints to type parameters.
- Use generic list collections.
- Use generic dictionary collections.
- Create custom generic collections.
- Create enumerable generic collections.

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, 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.

The following example shows how to create a generic class:

Creating a Generic Class

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

When you create an instance of your generic class, you specify the type you want to supply as a type parameter. For example, if you want to use your custom list to store objects of type **Coffee**, you would supply **Coffee** as the type parameter.

The following example shows how to instantiate a generic class:

Instantiating a Generic Class

```
CustomList<Coffee> clc = new CustomList<Coffee>;
Coffee coffee1 = new Coffee();
Coffee coffee2 = new Coffee();
clc.Add(coffee1);
clc.Add(coffee2);
Coffee firstCoffee = clc[0];
```

When you instantiate a class, every instance of **T** within the class is effectively replaced with the type parameter you supply. For example, if you instantiate the **CustomList** class with a type parameter of **Coffee**:

- The **Add** method will only accept an argument of type **Coffee**.
- The **Remove** method will only accept an argument of type **Coffee**.
- The indexer will always provide a return value of type **Coffee**.

Advantages of Generics

Advantages of Generics

Generic types offer three advantages over non-generic types:

- Type safety
- No casting
- No boxing and unboxing

The use of generic classes, particularly for collections, offers three main advantages over non-generic approaches: type safety, no casting, and no boxing and unboxing.

Type Safety

Consider an example where you use an **ArrayList** to store a collection of **Coffee** objects. You can add objects of any type to an **ArrayList**. Suppose a developer adds an object of type **Tea** to the collection. The code will build without complaint. However, a runtime exception will occur if the **Sort** method is called, because the collection is unable to compare objects of different types. Furthermore, when you retrieve an object from the collection, you must cast the object to the correct type. If you attempt to cast the object to the wrong type, an invalid cast runtime exception will occur.

The following example shows the type safety limitations of the **ArrayList** approach:

Type Safety Limitations for Non-Generic Collections

```
var coffee1 = new Coffee();
var coffee2 = new Coffee();
var tea1 = new Tea();
var arrayList1 = new ArrayList();
arrayList1.Add(coffee1);
arrayList1.Add(coffee2);
arrayList1.Add(tea1);
// The Sort method throws a runtime exception because the collection is not homogenous.
arrayList1.Sort();
// The cast throws a runtime exception because you cannot cast a Tea instance to a Coffee instance.
Coffee coffee3 = (Coffee)arrayList1[2];
```

As an alternative to the **ArrayList**, suppose you use a generic **List<T>** to store a collection of **Coffee** objects. When you instantiate the list, you provide a type argument of **Coffee**. In this case, your list is guaranteed to be homogenous, because your code will not build if you attempt to add an object of any other type. The **Sort** method will work because your collection is homogenous. Finally, the indexer returns objects of type **Coffee**, rather than **System.Object**, so there is no risk of invalid cast exceptions.

The following example shows an alternative to the **ArrayList** approach using the generic List<T> class:

Type Safety in Generic Collections

```
var coffee1 = new Coffee();
var coffee2 = new Coffee();
var tea1 = new Tea();
var genericList1 = new List<Coffee>();
genericList1.Add(coffee1);
genericList1.Add(coffee2);
// This line causes a build error, as the argument is not of type Coffee.
genericList1.Add(tea1);
// The Sort method will work because the collection is guaranteed to be homogenous.
arrayList1.Sort();
// The indexer returns objects of type Coffee, so there is no need to cast the return value.
Coffee coffee3 = genericList[1];
```

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. 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.

The following example shows the difference between generic and non-generic collections with regard to boxing and unboxing:

Boxing and Unboxing: Generic vs. Non-Generic Collections

```
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];
```

Constraining Generics

Constraining Generics

You can constrain type parameters in six ways:

- where T : <name of interface>
- where T : <name of base class>
- where T : U
- where T : new()
- where T : struct
- where T : class

In some cases, you may need to restrict the types that developers can supply as arguments when they instantiate your generic class. The nature of these constraints will depend on the logic you implement in your generic class. For example, if a collection class uses a property named **AverageRating** to sort the items in a collection, you would need to constrain the type parameter to classes that include the **AverageRating** property. Suppose the **AverageRating** property is defined by the **IBeverage** interface. To implement this restriction, you would constrain the type parameter to classes that implement the **IBeverage** interface by using the **where** keyword.

The following example shows how to constrain a type parameter to classes that implement a particular interface:

Constraining Type Parameters by Interface

```
public class CustomList<T> where T : IBeverage
{
}
```

You can apply the following six types of constraint to type parameters:

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.

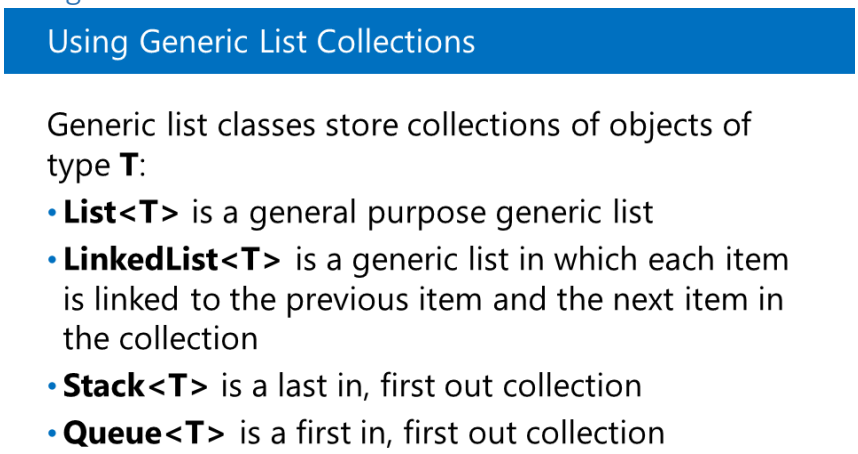
where T : U	The type argument must be, or derive from, the supplied type argument U.
where T : new()	The type argument must have a public default constructor.
where T : struct	The type argument must be a value type.
where T : class	The type argument must be a reference type.

You can also apply multiple constraints to the same class, as shown by the following example:

Apply Multiple Type Constraints

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

Using Generic List Collections



One of the most common and important uses of generics is in collection classes. Generic collections fall into two broad categories: generic list collections and generic dictionary collections. A generic list stores a collection of objects of type T.

The List<T> Class

The **List<T>** class provides a strongly-typed alternative to the **ArrayList** class. Like the **ArrayList** class, the **List<T>** class includes methods to:

- Add an item.
- Remove an item.
- Insert an item at a specified index.
- Sort the items in the collection by using the default comparer or a specified comparer.
- Reorder all or part of the collection.

The following example shows how to use the **List<T>** class.

Using the List<T> Class

```
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)
```

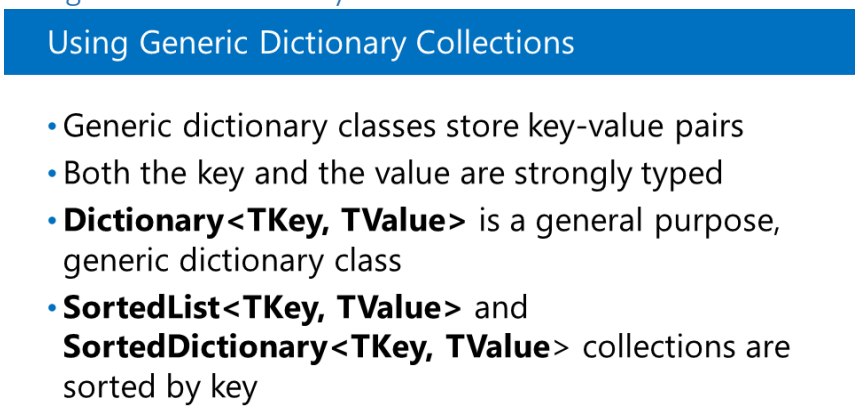
```
{
Console.WriteLine(coffeeBeverage);
}
Console.ReadLine ("Press Enter to continue");
```

Other Generic List Classes

The **System.Collections.Generic** namespace also includes various generic collections that provide more specialized functionality:

- The **LinkedList<T>** class provides a generic collection in which each item is linked to the previous item in the collection and the next item in the collection. Each item in the collection is represented by a **LinkedListNode<T>** object, which contains a value of type **T**, a reference to the parent **LinkedList<T>** instance, a reference to the previous item in the collection, and a reference to the next item in the collection.
- The **Queue<T>** class represents a strongly typed first in, last out collection of objects.
- The **Stack<T>** class represents a strongly typed last in, last out collection of objects.

Using Generic Dictionary Collections



Dictionary classes store collections of key value pairs. The value is the object you want to store, and the key is the object you use to index and retrieve the value. For example, you might use a dictionary class to store coffee recipes, where the key is the name of the coffee and the value is the recipe for that coffee. In the case of generic dictionaries, both the key and the value are strongly typed.

The Dictionary<TKey, TValue> Class

The **Dictionary<TKey, TValue>** class provides a general purpose, strongly typed 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.

The following example shows how to use the **Dictionary<TKey, TValue>** class:

Using the Dictionary<TKey, TValue> Class

```
// 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
```

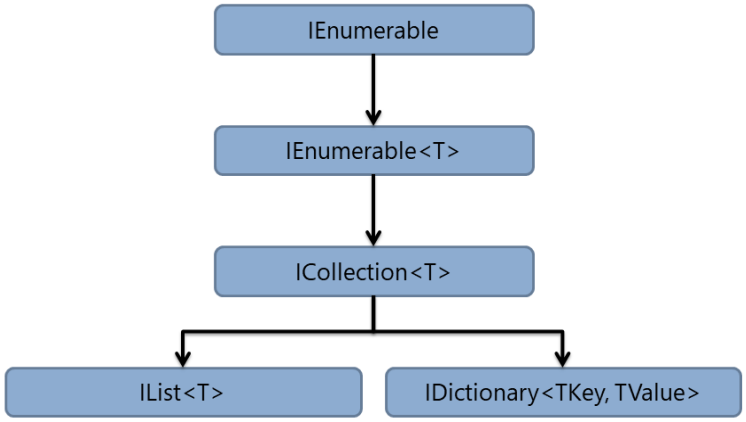
```
{
Console.WriteLine("The key \"CSM\" was not found");
}
// You can also use the indexer to change the value associated with a key.
coffeeCodes["IC"] = "Instant Coffee";
```

Other Generic Dictionary Classes

The **SortedList<TKey, TValue>** and **SortedDictionary<TKey, TValue>** classes both provide generic dictionaries in which the entries are sorted by key. The difference between these classes is in the underlying implementation:

- The **SortedList** generic class uses less memory than the **SortedDictionary** generic class.
- The **SortedDictionary** class is faster and more efficient at inserting and removing unsorted data.

Using Collection Interfaces



The **System.Collections.Generic** namespace provides a range of generic collections to suit various scenarios. However, there will be circumstances when you will want to create your own generic collection classes in order to provide more specialized functionality. For example, you might want to store data in a tree structure or create a circular linked list.

Where should you start when you want to create a custom collection class? All collections have certain things in common. For example, you will typically want to be able to enumerate the items in the collection by using a **foreach** loop, and you will need methods to add items, remove items, and clear the list. The picture on the slide shows that the .NET Framework provides a hierarchical set of interfaces that define the characteristics and behaviors of collections. These interfaces build on one another to define progressively more specific functionality.

The IEnumerable and IEnumerable<T> Interfaces

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. The **IEnumerable<T>** method defines a single method named **GetEnumerator()**. This method must return an object of type **IEnumerator<T>**. The **foreach** statement relies on this enumerator object to iterate through the collection.

The **IEnumerable<T>** interface *inherits* from the **IEnumerable** interface, which also defines a single method named **GetEnumerator()**. When an interface inherits from another interface, it exposes all the members of the parent interface. In other words, if you implement **IEnumerable<T>**, you also need to implement **IEnumerable**.

The ICollection<T> Interface

The **ICollection<T>** interface defines the basic functionality that is common to all generic collections. The interface inherits from **IEnumerable<T>**, which means that if you want to implement **ICollection<T>**, you must also implement the members defined by **IEnumerable<T>** and **IEnumerable**.

The **ICollection<T>** interface defines the following methods:

Name	Description
Add	Adds an item of type **T** to the collection.
Clear	Removes all items from the collection.

Contains	Indicates whether the collection contains a specific value.
CopyTo	Copies the items in the collection to an array.
Remove	Removes a specific object from the collection.

The **ICollection<T>** interface defines the following properties:

Name	Description
Count	Gets the number of items in the collection.
IsReadOnly	Indicates whether the collection is read-only.

The IList<T> Interface

The **IList<T>** interface defines the core functionality for generic list classes. You should implement this interface if you are defining a linear collection of single values. In addition to the members it inherits from **ICollection<T>**, the **IList<T>** interface defines methods and properties that enable you to use indexers to work with the items in the collection. For example, if you create a list named **myList**, you can use **myList[0]** to access the first item in the collection.

The **IList<T>** interface defines the following methods:

Name	Description
Insert	Inserts an item into the collection at the specified index.
RemoveAt	Removes the item at the specified index from the collection.

The **IList<T>** interface defines the following properties:

Name	Description
IndexOf	Determines the position of a specified item in the collection.

The IDictionary<TKey, TValue> Interface

The **IDictionary<TKey, TValue>** interface defines the core functionality for generic dictionary classes. You should implement this interface if you are defining a collection of key-value pairs. In addition to the members it inherits from **ICollection<T>**, the **IDictionary<T>** interface defines methods and properties that are specific to working with key-value pairs.

The **IDictionary<TKey, TValue>** interface defines the following methods:

Name	Description
Add	Adds an item with the specified key and value to the collection.
ContainsKey	Indicates whether the collection includes a key-value pair with the specified key.
GetEnumerator	Returns an enumerator of **KeyValuePair<TKey, TValue>** objects.
Remove	Removes the item with the specified key from the collection.
TryGetValue	Attempts to set the value of an output parameter to the value associated with a specified key. If the key exists, the method returns true. If the key does not exist, the method returns false and the output parameter is unchanged.

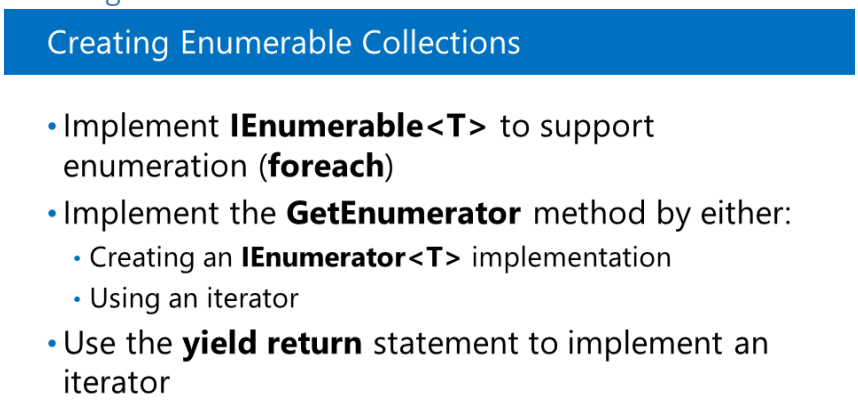
The **IDictionary<TKey, TValue>** interface defines the following properties:

Name	Description
Item	Gets or sets the value of an item in the collection, based on a specified key. This property enables you to use indexer notation, for example **myDictionary[myKey] = myValue**.

Keys	Returns the keys in the collection as an **ICollection<T>** instance.
Values	Returns the values in the collection as an **ICollection<T>** instance.

Reference Links: For comprehensive information and examples of all of the generic interfaces covered in this topic, see the System.Collections.Generic Namespace page at https://aka.ms/moc-20483c-m4-pg2.

Creating Enumerable Collections



To enumerate over a collection, you typically use a **foreach** loop. The **foreach** loop exposes each item in the collection in turn, in an order that is appropriate to the collection. The **foreach** statement masks some of the complexities of enumeration. For the **foreach** statement to work, a generic collection class must implement the **IEnumerable<T>** interface. This interface exposes a method, **GetEnumerator**, which must return an **IEnumerator<T>** type.

The IEnumerator<T> Interface

The **IEnumerator<T>** interface defines the functionality that all enumerators must implement.

The **IEnumerator<T>** interface defines the following methods:

Name	Description
MoveNext	Advanced the enumerator to the next item in the collection.
Reset	Sets the enumerator to its starting position, which is before the first item in the collection.

The **IEnumerator<T>** interface defines the following properties:

Name	Description
Current	Gets the item that the enumerator is pointing to.

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.

When you create an enumerator, you must define:

- Which item the enumerator should treat as the first item in the collection.
- In what order the enumerator should move through the items in the collection.

The IEnumerable<T> Interface

The **IEnumerable<T>** interface defines a single method named **GetEnumerator**. This returns 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.

The following example shows a custom collection class that implements a default enumerator, together with an alternative enumerator that enumerates the collection in reverse order:

Default and Alternative Enumerators

```
class CustomCollection<T> : IEnumerable<T>
{
public IEnumerator<T> Backwards()
{
// This method returns an alternative enumerator.
// The implementation details are not shown.
}
#region IEnumerable<T> Members
public IEnumerator<T> GetEnumerator()
{
// This method returns the default enumerator.
// The implementation details are not shown.
}
#endregion
#region IEnumerable Members
IEnumerator IEnumerable.GetEnumerator()
{
// This method is required because IEnumerable<T> inherits from IEnumerable
throw new NotImplementedException();
}
#endregion
}
```

The following example shows how you can use a default enumerator or an alternative enumerator to iterate through a collection:

Enumerating a Collection

```
CustomCollection<Int32> numbers = new CustomCollection<Int32>();
// Add some items to the collection.
// Use the default enumerator to iterate through the collection:
foreach (int number in numbers)
{
// …
}
// Use the alternative enumerator to iterate through the collection:
foreach(int number in numbers.Backwards())
{
// …
}
```

Implementing the Enumerator

You can provide an enumerator by creating a custom class that implements the **IEnumerator<T>** interface. 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. The best way to understand iterators is to start with a simple example.

The following example shows how you can use an iterator to implement an enumerator:

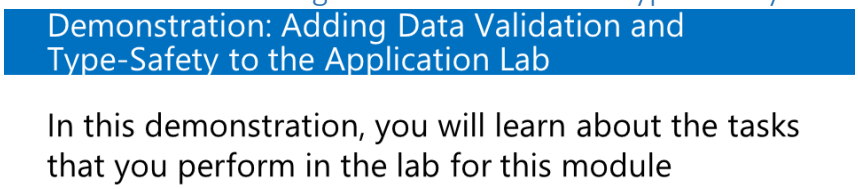
Implementing an Enumerator by Using an Iterator

```
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();
}
```

```
}
```

The example shows a custom generic collection class that uses a **List<T>** instance to store data. The **List<T>** instance is populated by the **FillList** method. When the **GetEnumerator** method is called, a **foreach** loop enumerates the underlying collection. Within the **foreach** loop, a **yield return** statement is used to return each item in the collection. It is this **yield return** statement that defines the iterator—essentially, the **yield return** statement pauses execution to return the current item to the caller before the next element in the sequence is retrieved. In this way, although the **GetEnumerator** method does not appear to return an **IEnumerator** type, the compiler is able to build an enumerator from the iteration logic that you provided.

Demonstration: Adding Data Validation and Type-Safety to the Application Lab

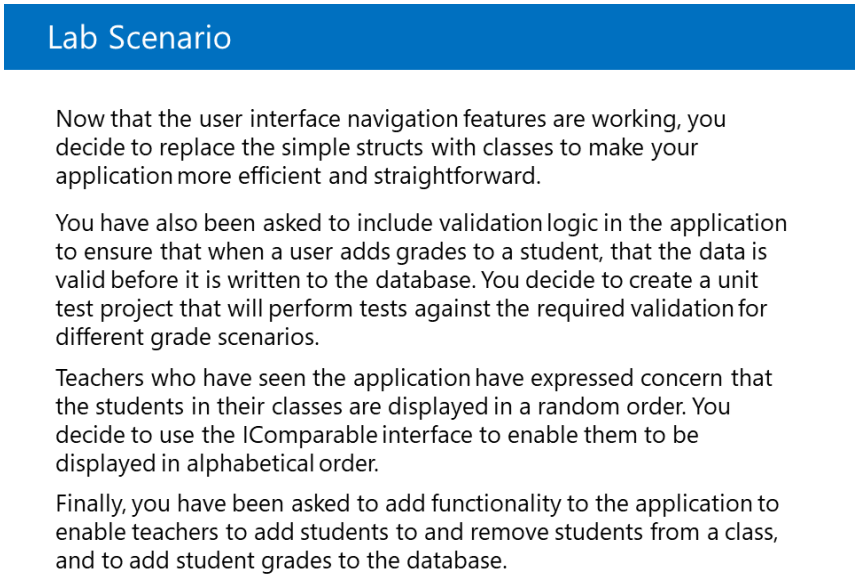


In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration Steps

You will find the steps in the **Demonstration: Adding Data Validation and Type-Safety to the Application Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_DEMO.md.

Lab Scenario



Lab: Adding Data Validation and Type-Safety to the Application

Lab: Adding Data Validation and Type-Safety to the Application

- Exercise 1: Implementing the Teacher, Student, and Grade Structs as Classes
- Exercise 2: Adding Data Validation to the Grade Class
- Exercise 3: Displaying Students in Name Order
- Exercise 4: Enabling Teachers to Modify Class and Grade Data

Estimated Time: 75 minutes

Scenario

Now that the user interface navigation features are working, you decide to replace the simple structs with classes to make your application more efficient and straightforward.

You have also been asked to include validation logic in the application to ensure that when a user adds grades to a student, that the data is valid before it is written to the database. You decide to create a unit test project that will perform tests against the required validation for different grade scenarios.

Teachers who have seen the application have expressed concern that the students in their classes are displayed in a random order. You decide to use the IComparable interface to enable them to be displayed in alphabetical order.

Finally, you have been asked to add functionality to the application to enable teachers to add students to and remove students from a class, and to add student grades to the database.

Objectives

After completing this lab, you will be able to:

- Create classes.
- Write data validation code and unit tests.
- Implement the **IComparable** interface.
- Use generic collections.

Lab Setup

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_LAK.md.

Module Review and Takeaways

Module Review and Takeaways

- Review Questions

In this module, you have learned how to work with classes, interfaces, and generic collections in Visual C#.

Review Questions

1. Which of the following types is a reference type?

( )Boolean
( )Byte
( )Decimal
( )Int32
( )Object

2. Which of the following types of member CANNOT be included in an interface?

( )Events
( )Fields
( )Indexers
( )Methods
( )Properties

3. 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?

( )public class DoubleEndedQueue<T> : IEnumerable<T>
( )public class DoubleEndedQueue<T> : ICollection<T>
( )public class DoubleEndedQueue<T> : IList<T>
( )public class DoubleEndedQueue<T> : IList<T>, IEnumerable<T>
( )public class DoubleEndedQueue<T> : IDictionary<TKey,TValue>