Structs and Interfaces

Objectives

- Create structs as lightweight value-type objects.
- Create interfaces as contracts for classes to fulfill.
- Instantiate interfaces.
- Pass interfaces to methods.
- Extend interfaces to provide specialization.
- Use the keywords *is* and *as* to test the implementation of interfaces.

Alternatives to Classes

The principal method for defining new types in C# is to create a class. Classes offer the full array of support for creating types, and much of your work as a C# developer will be in designing and implementing classes.

There are times, however, when you will want to create special types to meet extraordinary requirements. This chapter focuses on two such occasions.

You may, from time to time, need to create a lightweight value type (rather than a reference type). For this, you may choose to create a *struct*, rather than a class. There are limitations with structs (discussed in the next section) but if what you need is a small value type, structs may be the answer.

More important, you will often want to define a contract that other classes must fulfill. For example, you may want to define what methods an object must have if it is to be stored in your database. To create such a contract, define an *interface*.

This chapter provides examples of how to use each of these alternative types.

Structs

A struct is a lightweight user-defined value type. They are similar to classes and may have properties, methods, constructors, fields, operators, nested types, and indexers. They are different from classes in some very important ways, including:

- Structs do not support inheritance.
- Structs are value types, not reference types.
- Structs do not support user-defined default constructors.
- There is no initialization.

TIP:	Structs are sealed, meaning that you may not inherit from them, nor may a struct inherit from any class except *object*.

Defining Structs

You define a struct much as you do a class. The syntax is as follows:

```
[attributes] [access-modifiers] struct identifier [
:interface-list] { struct members }
```

Attributes are discussed in the advanced portion of this course.

Access modifiers include public and private. Typically, structs are public.

The key word struct is followed by an identifier—the struct name is like a class name.

The interface list shows which interfaces the struct implements. Interfaces are described later in this chapter.

The struct members define the struct, much like the class members define the class.

Try It Out!

See ***structs.sln***

To see how to work with structs, you'll create a simple application that defines a struct and manipulates instances of the struct.

1. Create a new console application named **structs**.

2. Create a struct named **Book** and give it three member fields: *bookName* and *isbn*, both of which are strings. Also give it a public member *salesRank* of type int. It is not uncommon to have public fields in a struct, though some might argue that it is not a good programming practice.

```
namespace structs
{
   public struct Book
   {
      private string bookName;
      private string isbn;
      public int salesRank;
```

3. Provide a constructor that takes a name and an isbn and assigns them to the member fields. Note that your constructor must specifically set a value for the salesRank. There can be no initializer in a struct, so you must set this value in the constructor.

```
public Book(string name, string isbn)
{
   bookName = name;
   this.isbn = isbn;
   salesRank = -1;
}
```

4. Create properties for the two private fields.

```
public string BookName
{
   get { return bookName; }
   set { bookName = value; }
}

public string ISBN
{
   get { return isbn; }

}
```

5. Override the ToString() method. Note that while structs do not support inheritance (they are sealed) they do still derive from object, and may override methods of object.

```
public override string ToString()
{
   return (String.Format("{0} ISBN: {1} Rank: {2}",
      bookName, isbn, salesRank));
}
```

6. Create a test class to test your new struct. In the Run() method, instantiate two struct objects.

```
Book progASP =
   new Book("Programming ASP.NET","0596001711");
Book cSharp =
   new Book("Programming C#", "0596001177");
```

7. Assign the following to the public field.

```
progASP.salesRank = 12;
cSharp.salesRank = 2;
```

8. Use the public field to access information about the book.

```
Console.WriteLine ("{0} has a sales rank of {1}",
   cSharp.BookName, cSharp.salesRank);
```

9. Use the overridden ToString method.

```
Console.WriteLine(progASP);
```

The output of this program is shown in Figure 1.

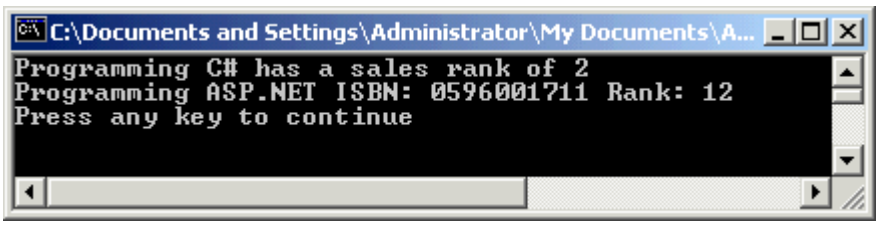

Figure 1. Testing the struct.

Structs Are Value Types

The Book struct you've created creates value objects. If you pass a Book to a method, a copy is made. You can prove this to yourself by creating a new method, *PassByValue* that takes a Book object as a parameter:

```
public void PassByValue(Book theBook)
{
   theBook.salesRank = 20;
   Console.WriteLine("In the book: {0}", theBook);
}
```

This method modifies the Book object passed in; it sets the salesRank to 20 and then displays the information about the book.

Call this method from your test program, displaying the book both before and after the call:

```
Console.WriteLine(progASP);
Console.WriteLine("\nPassing progASP to PassByValue...");
PassByValue(progASP);
Console.WriteLine("Back from PassByValue: {0}", progASP);
```

The results are shown in Figure 2. You can see that the Rank was set within the method, but when you return to Main the value is not changed. This is consistent with pass by value semantics.

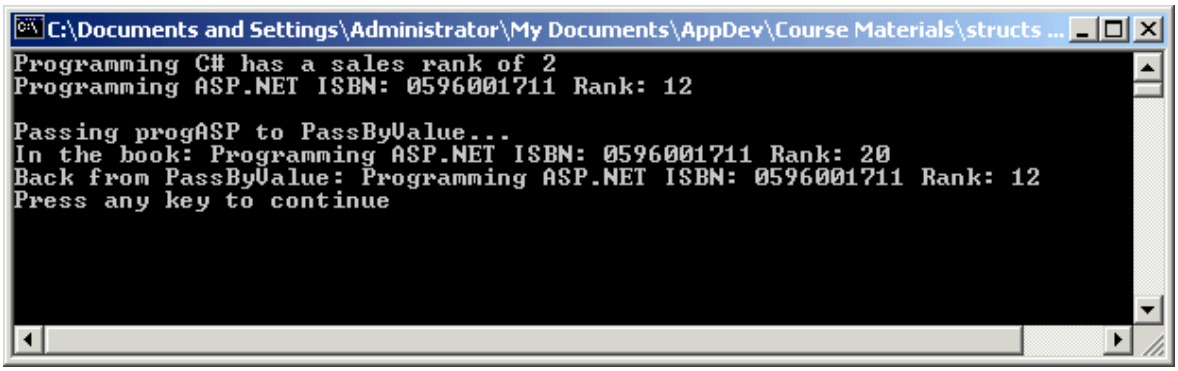

Figure 2. Passing by value.

If you do need to pass a struct by reference, you do so as you would with an intrinsic type: using the ref keyword. Add another method, *PassByRef* that takes a Book as a reference:

```
public void PassByRef(ref Book theBook)
{
   theBook.salesRank = 20;
   Console.WriteLine("In the book: {0}", theBook);
}
```

This method is identical to *PassByValue* except that the Book type is preceded by the keyword *ref*. Remember to use the *ref* keyword when you invoke the method as well:

```
Console.WriteLine("\nPassing progASP to PassByRef...");
PassByRef(ref progASP);
Console.WriteLine("Back from PassByRef: {0}", progASP);
```

The output is shown in Figure 3. You can see that using the keyword ref avoided the copy and the original object is now modified.

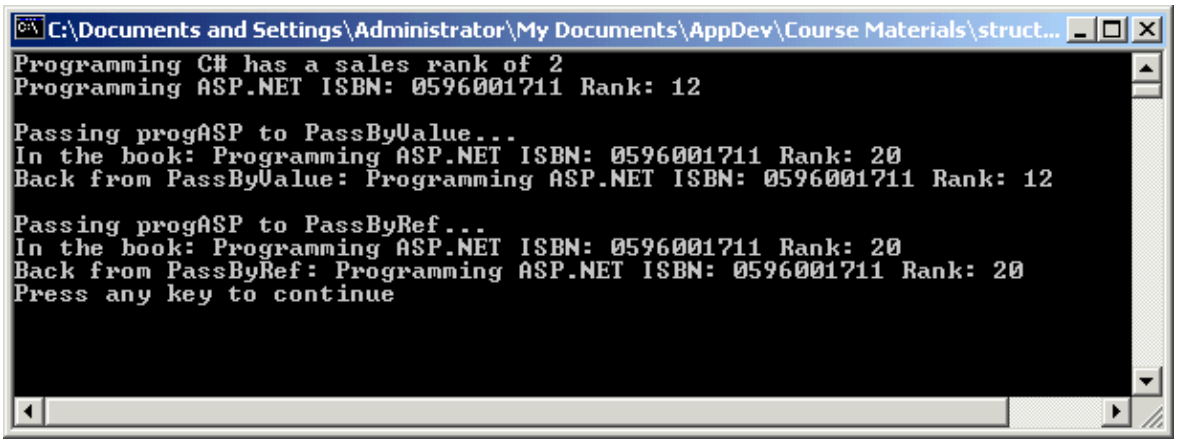

Figure 3. Passing with the ref keyword.

Be Careful with Structs

Because structs are value objects they can degrade performance when you store them in collections. Most .NET collections expect to store reference objects. When you pass in a struct the struct must be boxed, and there is overhead in boxing an object. This can degrade performance when you are storing many structs. This lightweight object can actually hurt your program's performance rather than helping it!

Interfaces

An interface is a contract. When a class implements the interface it must support the methods, properties, and events of the interface. The interface describes the contract, and the class implements that contract.

The designer of a class may choose not to implement any interfaces, or the designer may choose to implement one or more interfaces. That is entirely a design decision, but if an interface *is* implemented, then it must be implemented in full.

The formal definition of an interface is as follows:

```
[attributes]
[access-modifier] interface identifier
[:base-list]
{interface body}
```

Attributes are an advanced topic, covered in future chapters.

The access modifier is typically public. An interface can be declared with any visibility, but interface members must all be public.

The keyword interface is followed by the interface name. It is a convention to have that name begin with the letter I, as in IStorable, ITransmittable, and so forth.

The base list includes the list of interfaces that this interface extends. You'll see more about extending interfaces later in this chapter.

The interface body defines the methods, properties, and events that must be implemented by the class fulfilling the contract.

Try It Out!

See ***Interfaces.sln***

To get started, you'll create a simple interface: IInvestment.

1. Open a new console application and call it **Interfaces**.

2. Create a public interface named **IInvestment** and give it two methods as shown here.

```
public interface Iinvestment
{
   void Invest(float amount);
   void DisplayCurrentValue();
}
```

3. Notice that the Interface methods do not include implementation. The return type and parameters are listed, but there is no body; the definition is followed by a semicolon.

4. Also notice that the methods have no access modifiers. All interface methods must be public.

5. You have now created a contract. To fulfill that contract, you need a class that implements the interface. Create a Stock class and indicate that it implements the interface using the colon operator:

```
public class Stock : IInvestment
{
```

6. The colon followed by the name of the interface indicates that the class implements the interface. This looks very similar to derivation, and it is up to the compiler to determine that Stock does not inherit from IInterface, but rather fulfills the interface.

7. You are free to give the implementing class any fields and methods you like, so long as you *also* include all the members required by the interface definition. Start by adding three private members:

```
private string name;
private float price;
private float numShares;
```

8. Initialize these values in the constructor.

```
public Stock(
   string name,
   float price,
   float numShares)
 {
   this.name = name;
   this.price = price;
   this.numShares = numShares;
}
```

9. Provide public read-only properties for the name and number of shares.

```
public string Name
{
   get
   {
      return name;
   }
}
public float NumShares
{
   get
   {
      return numShares;
   }
}
```

10. Finally, it is time to implement the two methods required by the interface. The first is *Invest*, which must return void and take a float indicating the amount to invest. In a real program you might implement this by actually purchasing the stock and updating database records. For this simplified example, you'll just set a member variable:

```
public void Invest (float amt)
{
   this.numShares += (amt/price);
}
```

11. Similarly, you must implement DisplayCurrentValue; a method taking no parameters and returning void.

```
public void DisplayCurrentValue()
{
   Console.WriteLine("{0}: totalValue: {1}",
      name, price*numShares);
}
```

12. Test this program by creating an instance of Stock and setting its values. You can then call methods defined in the interface to interact with the stock as an investment.

```
public class Tester
{
   public void Run()
   {
      Stock myStock = new Stock("Liberty", 5.7f, 100f);
      myStock.Invest(5200f);
      myStock.DisplayCurrentValue();
      Console.WriteLine("{0} has {1} shares",
         myStock.Name, myStock.NumShares);
```

The results are shown in Figure 4.

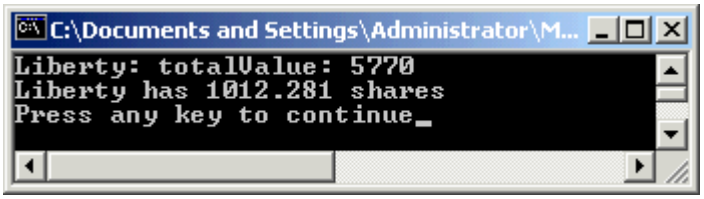

Figure 4. Stock class implements Iinvest.

13. To see the flexibility of the interface, create a second class, OilWell that will also implement this interface. An OilWell has different private member variables, and it can have different methods than Stock, but will also implement the required methods. Of course, the details of how an OilWell implements the Invest method may differ from how it is implemented by Stock.

```
public class OilWell : Iinvestment
{
   private string wellID;
   private float investment = 0;

   public OilWell(string wellID)
   {
      this.wellID = wellID;
   }

   public string WellID
   {
      get { return wellID; }
   }

   public void Invest(float amount)
   {
      this.investment = amount;
   }

   public void DisplayCurrentValue()
   {
      Console.WriteLine("{0}: has a current value of {1}",
         wellID, investment);
   }
}
```

14. Notice that in this case, the OilWell invests by setting its investment field to the total value of the amount invested, while a stock implements *Invest* by incrementing the number of shares, computed by dividing the amount invested by the current price.

15. Add test code to your test method to instantiate an OilWell and invest in it.

```
OilWell myWell = new OilWell("J3752");
myWell.Invest(10000f);
myWell.DisplayCurrentValue();
```

The results are shown in Figure 5.

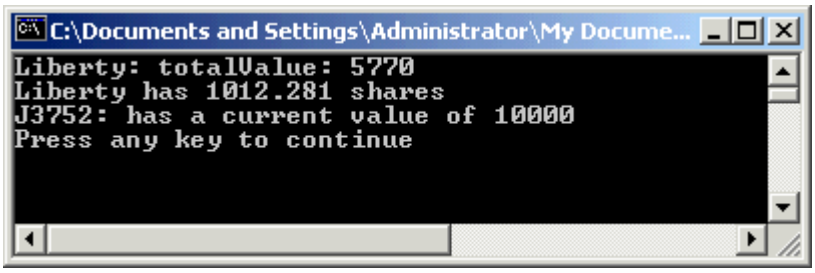

Figure 5. A second class implements Iinvest.

Instantiating the Interface

Once you have classes that have implemented the interface, you can create instances of the interface itself. This allows you to interact with the implementing classes polymorphically. That is, you can interact *through* the interface and access all of the members of the interface, without regard to which class has implemented it.

This ability to work with an interface generically allows you to create methods that expect an interface object, and pass in an instance of any object that implements that interface. You can see this by adding code to the test method in the example just reviewed:

```
IInvestment myInvestment = myStock;
Console.WriteLine(
   "\nCalling myInvestment.DisplayCurrentValue()");
myInvestment.DisplayCurrentValue();
```

You have instantiated myInvestment, an object of type IInvestment, which you have initialized to refer to the object myStock. Since myStock implements IInvestment, this is a reasonable thing to do. You are now free to call any IInvestment method such as DisplayCurrentValue, and you will call that method on the myStock instance, as shown in Figure 6.

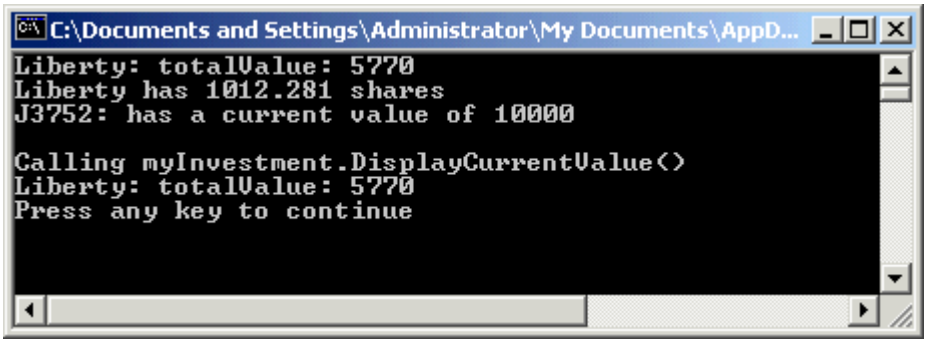

Figure 6. Instantiating the interface.

You can reassign that same reference to IInvestment to point to the OilWell object:

```
myInvestment = myWell;
Console.WriteLine(
   "\nCalling myInvestment.DisplayCurrentValue()");
myInvestment.DisplayCurrentValue();
```

The result of adding these lines is shown in Figure 7. You can see here that myInvestment is first assigned to myStock, and then reassigned to myWell.

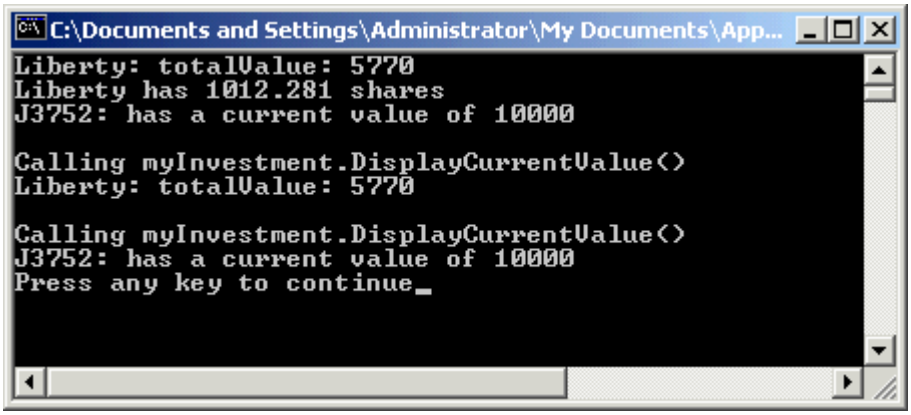

Figure 7. Reassigning the reference to Iinvestment.

Implementing More than One Interface

While a class can only inherit from (at most) one other class, it can implement any number of interfaces. This allows you to define small sets of functionality as interfaces, and then implement classes that mix and match that functionality. To see this, extend the previous example to add a second interface.

Try It Out!

*See **Interfaces2.sln***

In this next example you will add a second interface, ITaxable, and define classes that implement more than one interface.

1. Reopen the previous example.

2. Add a second interface, ITaxable, to support investments on which you will pay tax.

```
interface ITaxable
{
   float ComputeTaxes();
}
```

3. Modify Stock to implement both interfaces.

```
public class Stock : IInvestment, Itaxable
```

4. Add a method to Stock to implement Itaxable.

```
public float ComputeTaxes()
{
   return (price * numShares) * 0.3f;
}
```

5. Do not modify OilWell. Just because stocks are both Investments and Taxable does not mean that all investments are taxable.

```
// no change
public class OilWell : IInvestment
```

6. Modify the test program to create a stock and test its implementation of the interface.

```
public void Run()
{
   Stock myStock = new Stock("Liberty", 5.7f, 100f);
   myStock.Invest(5200f);
   myStock.DisplayCurrentValue();
   Console.WriteLine("{0} has {1} shares",
      myStock.Name, myStock.NumShares);
   Console.WriteLine("Taxes on my stock investment: {0}",
      myStock.ComputeTaxes());
```

The results are shown in Figure 8.

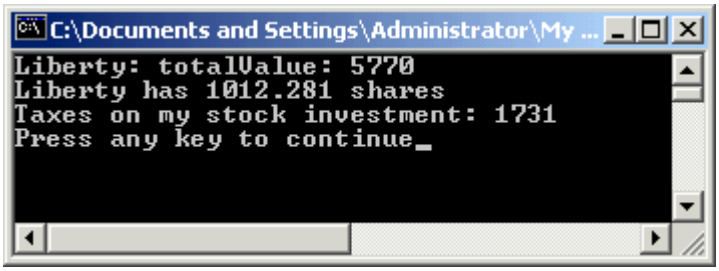

Figure 8. Implementing two interfaces.

7. Try calling ComputeTaxes on the OilWell.

```
OilWell myWell = new OilWell("J3752");
myWell.Invest(10000f);
myWell.DisplayCurrentValue();
Console.WriteLine("Taxes on my stock investment: {0}",
   myWell.ComputeTaxes());
```

When you try to compile this the compile fails with the following message:

```
Interfaces2.OilWell does not contain a definition for
'ComputeTaxes'
```

Oops; OilWell did not implement the ITaxable interface.

Extending Interfaces

Just as you can derive a new class from an existing class, you can extend an existing interface. This allows you to specialize the requirements of an interface, and allow classes to choose the level of specialization they will implement.

Try It Out!

See ***Interfaces3.sln***

To see how this works, reopen the previous example. This time you'll add an extension to one of the interfaces you created in the previous example.

1. Reopen the previous example.

2. Create a new interface, IShelterable that extends ITaxable. This supports investments that are not only taxable, but eligible to be used as a tax shelter.

```
interface IShelterable : ITaxable
{
   float TaxableAmount { get; }
   void ReduceTaxImpact();
}
```

Notice that extending the interface uses the same syntax as deriving from a class. That is, you write the name of the new interface, followed by a colon, followed by the name of the base interface.

This new interface, IShelterable, adds a property and a new method. Notice also that the property is not implemented it is just defined.

3. Modify OilWell to implement Ishelterable.

```
public class OilWell : IInvestment, IShelterable
```

4. You will have to implement not only the methods of IShelterable, but also the methods of ITaxable, the interface IShelterable extends. To accomplish this, start by adding a member field *taxableAmount* to the OilWell:

```
private string wellID;
private float investment = 0;
private float taxableAmount;
```

5. Implement the method of Itaxable.

```
public float ComputeTaxes()
{
   return taxableAmount * 0.3f;
}
```

6. Implement the method of Ishelterable.

```
public void ReduceTaxImpact()
{
   taxableAmount -= taxableAmount * .97f;
}
```

7. Modify the test program to test the Shelterable interface.

```
public class Tester
{
   public void Run()
   {
      Stock myStock = new Stock("Liberty",5,100);
      myStock.Invest(5200);
      myStock.DisplayCurrentValue();
      Console.WriteLine(
         "Taxes on my stock investment: {0}",
         myStock.ComputeTaxes());

      OilWell myWell = new OilWell("Well 15");
      myWell.Invest(20000);
      Console.WriteLine(
         "Taxes on my well: {0}",myWell.ComputeTaxes());
      myWell.ReduceTaxImpact();
```

```
            Console.WriteLine(
                "Taxes on my well reduced to: {0}",
                myWell.ComputeTaxes());
        }

        public static void Main()
        {
            Tester t = new Tester();
            t.Run();
        }
}
```

The result of running this code is shown in Figure 9. You can see that the OilWell is able to reduce its taxes by implementing IShelterable!

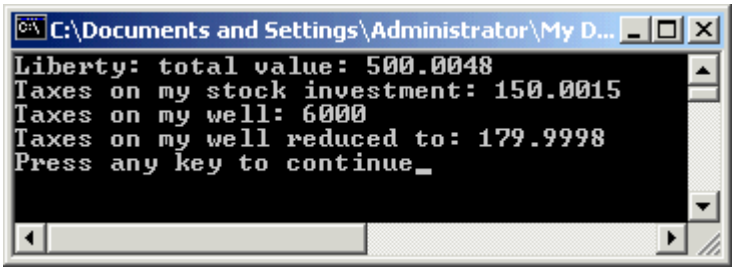

Figure 9. Extending an interface.

Testing an Implementation

It is possible to create a method that takes a Taxable object (that is, an object that implements ITaxable) and pass to that method both objects that implement ITaxable and also objects that implement IShelterable. That is, wherever you can use an interface, you can use an extension to that interface.

Within such a method you may need to test whether the object you have actually implements one or another extension. You can test an interface with the *is* keyword. The keyword is will return true if the object implements the interface you are testing and false if it does not.

Try It Out!

*See **Interfaces3.sln***

Modify the previous example by adding a new method, *Shelter*.

1. Reopen the previous example.

2. Add a new method, Shelter that takes an ITaxable object. This method takes a second parameter, *identifier* as a string.

```
public void Shelter(
   ITaxable investment, string identifier)
 {

    Console.WriteLine("Taxes on {0}: {1}",
       identifier,
       investment.ComputeTaxes());
```

3. Test the investment, and if it is shelterable then apply the shelter, otherwise indicate that by writing to the console.

```
if (investment is IShelterable)
 {
    IShelterable shelterable = (IShelterable) investment;
    shelterable.ReduceTaxImpact();
    Console.WriteLine(
       "Taxes reduced to: {0}",
       shelterable.ComputeTaxes());
 }
 else
    Console.WriteLine ("{0} is not shelterable!",
       identifier);
```

4. Add test code to test this method.

```
OilWell myWell = new

Shelter(myWell,myWell.WellID);


Stock myStock = new Stock("Liberty",5,100);
 myStock.Invest(5200);
myStock.DisplayCurrentValue();
Shelter(myStock,myStock.Name);
```

When you run the program, the results are as shown in Figure 10. The stock is not shelterable, but the OilWell is.

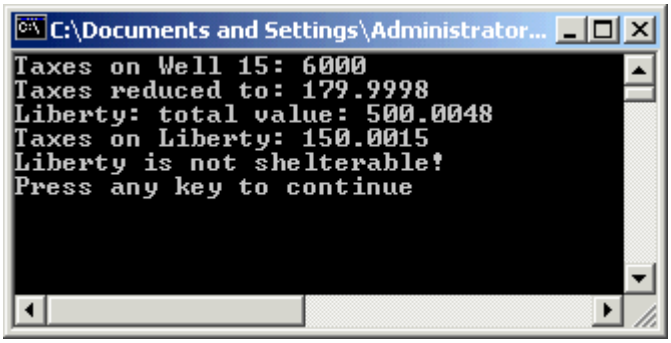

Figure 10. Passing interfaces to methods.

Using the *as* Keyword

While the is keyword works, it is not efficient to test the interface and then cast it. You can combine these two steps by using the *as* keyword. The keyword *as* casts the interface, and if the cast fails it returns null.

*See **Interfaces5.sln*** You can see the difference with a small change to the example just completed.

1. Reopen the project you just completed.
2. Modify the Shelterable method to use *as* rather than *is*.

```
public void Shelter(
   ITaxable investment,
   string identifier)
{

   Console.WriteLine("Taxes on {0}: {1}",
      identifier,
      investment.ComputeTaxes());

   IShelterable shelterable =
      investment as IShelterable;

   if (shelterable != null)
   {
      shelterable.ReduceTaxImpact();
      Console.WriteLine("Taxes reduced to: {0}",
         shelterable.ComputeTaxes());
   }
   else
      Console.WriteLine ("{0} is not shelterable!",
         identifier);
}
```

3. Notice that in this example you cast the investment to a Shelterable using the keyword *as*, and then test to see if the cast interface is null.

```
IShelterable shelterable =
   investment as IShelterable;

if (shelterable != null)
{
```

This is more efficient than first testing with *is* and then casting. The results are identical, as shown in Figure 11.

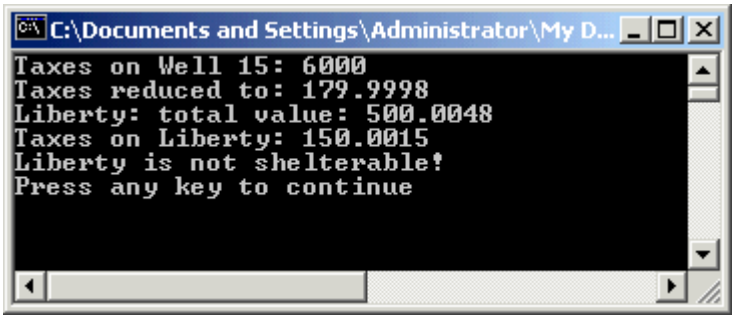

Figure 11. Using *as* rather than *is*.

Summary

- Structs are lightweight user-defined value types.
- Structs may have properties, methods, constructors, fields, operators, nested types, and indices.
- Structs are sealed and do not support user-defined default constructors or member initialization.
- Interfaces create a contract.
- You may instantiate an interface, and you may make an interface be a parameter to a method.
- A class may implement zero, one, or more interfaces.
- An interface may be *extended* much as a class is derived from.
- You may test whether an object implements a particular interface using either the is or the as keyword.

(Review questions and answers on the following pages.)

Questions

1. What is the difference between a struct and a class?
2. What is an interface?
3. Can a class implement more than one interface?
4. How do you extend an interface?
5. How do you test if an instance of a class implements an interface?
6. What is the difference between the keywords *is* and *as*?

Answers

1. What is the difference between a struct and a class?
   A struct is a value type. Structs are sealed and they do not support user-defined constructors. Structs do not support initialization.

2. What is an interface?
   An interface defines a contract that will be fulfilled (implemented) by a class. The interface defines the members (methods, properties, events, etc.) that the implementing class must provide.

3. Can a class implement more than one interface?
   Yes, a class may implement no interfaces, a single interface, or any other number of interfaces.

4. How do you extend an interface?
   You extend an interface much like you derive from a base class, using the colon operator (:). Classes implementing the extended interface must fulfill the contract of the base interface as well.

5. How do you test if an instance of a class implements an interface?
   You can use the is keyword that returns a Boolean value, or you can cast using the as keyword, which returns null if the cast fails.

6. What is the difference between the keywords *is* and *as*?
   The keyword is provides a test that returns a Boolean, but does not actually make the cast. The keyword as performs the cast, but if the cast fails, it returns null.

Lab 8: Structs and Interfaces

Lab 8 Overview

In this lab you'll learn how to create structs and interfaces.

To complete this lab, you'll need to work through two exercises:

- Creating Structs
- Implementing Interfaces

Each exercise includes an "Objective" section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

Creating Structs

Objective

In this exercise, you'll create a struct to represent a simple CD object that might be used to keep track of popular music CDs in your inventory. You will explore whether CDs are value or reference objects, and see that CDs provide much the same kind of interface as classes do.

Things to Consider

- Is a struct a reference or a value type?
- Why would you use structs rather than classes?
- What are the limitations of structs?

Step-by-Step Instructions

1. Open the project named **Structs_Starter.sln** in the Structs_Starter folder.

2. Create a structure for a CD that might contain information about the CDs name, artist, ISBN, and sales rank.

```
public struct CD
{
   private string cdName;
   private string artist;
   private string isbn;
   private int salesRank;
```

3. Create a constructor to set all the values—set the sales rank to –1.

```
public CD(string name, string isbn, string artist)
{
   cdName = name;
   this.artist = artist;
   this.isbn = isbn;
   salesRank = -1;
}
```

4. Create properties for the four members.

```
public string CDName
{
   get { return cdName; }
   set { cdName = value; }
}

public string ISBN
{
   get { return isbn; }
   set { isbn = value; }
}

public string Artist
{
   get { return artist; }
   set { artist = value; }
}

public int SalesRank
{
   get { return salesRank; }
   set { salesRank = value; }
}
```

5. Override ToString to display the name, artist, ISBN, and rank.

```
public override string ToString()
      {
         return (String.Format(
          "{0} by {1}. ISBN: {2} Rank: {3}",
            cdName, artist, isbn, salesRank));
      }
   }
```

6. Create a public method within Tester that takes a CD object by value and sets the sales rank and artist. Display the value to the console:

```
public void PassByValue(CD theCD)
{
   theCD.SalesRank = 20;
   theCD.Artist = "D. Brubeck";
   Console.WriteLine("In PassByValue with : {0}", theCD);
}
```

7. Create a public method that takes a CD object by reference and sets the sales rank and the artist. Display the values to the console.

```
public void PassByRef(ref CD theCD)
{
   theCD.SalesRank = 20;
   theCD.Artist = "D. Brubeck";
   Console.WriteLine("In PassByRef with: {0}", theCD);
}
```

8. Create a public method, Run that instantiates two CD objects.

```
public void Run()
{
   CD TakeFive =
      new CD("Take Five", "B00005J96U", "Dave Brubeck");
   CD Rhapsody =
      new CD("Rhapsody in Blue", "B0000026FG",
       "Gershwin");
```

9. Set the sales ranks for the two CDs and display one of them in detail.

```
TakeFive.SalesRank = 266;
Rhapsody.SalesRank = 712;
Console.WriteLine ("{0} has a sales rank of {1}",
   TakeFive.CDName, TakeFive.SalesRank);
Console.WriteLine(TakeFive);
```

10. Pass one of the CDs to the method that takes a CD struct by value. Display its contents when you return:

```
Console.WriteLine(
   "\nPassing TakeFive to  PassByValue...");
PassByValue(TakeFive);
Console.WriteLine("Back from PassByValue: {0}", TakeFive);
```

11. Pass one of the CDs to the method that takes a CD struct by reference. Display its contents when you return:

```
Console.WriteLine("\nPassing TakeFive to PassByRef...");
   PassByRef(ref TakeFive);
Console.WriteLine("Back from PassByRef: {0}", TakeFive);
```

12. Compile and run the program. It should look like Figure 12.

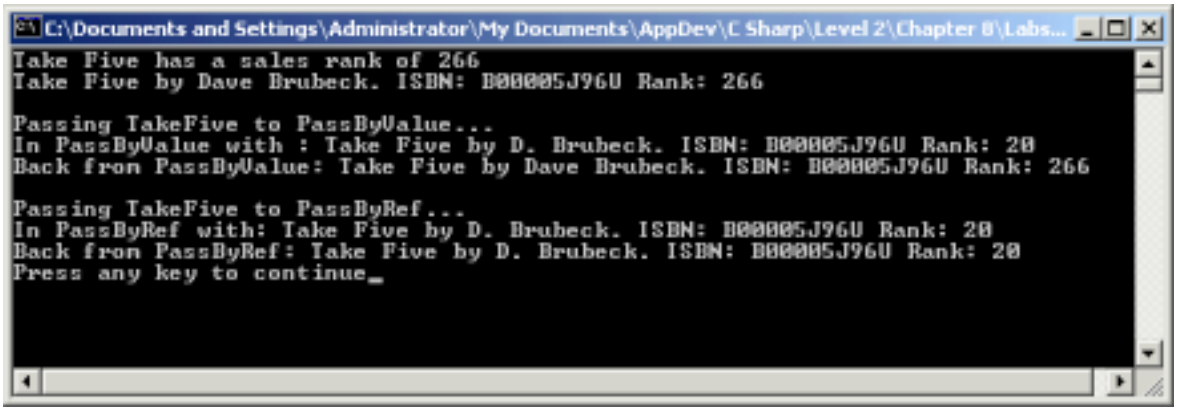

Figure 12. Final exercise results.

Implementing Interfaces

Objective

In this exercise, you'll create and extend interfaces, and you'll create classes that implement the interfaces.

Things to Consider

- What is the purpose of an interface?
- Why might you extend an interface?
- What must a class do to implement the interface?

Step-by-Step Instructions

1. Open the project named **Interfaces_Starter.sl**n in the Interfaces_Starter folder.

2. Create an interface named **IShelfable**. This will represent items that might be on a shelf in your store. The interface requires that implementing classes provide a method PutOnShelf that returns void and takes no parameters, and a read-only property that provides the classification for the shelfable object:

```
public interface IShelfable
{
   void PutOnShelf();
   string Classification { get; }
}
```

3. Create an interface ISellable for any item that might be sold. Provide a property for the selling price:

```
public interface ISellable
{
   float Price { get; set; }
}
```

4. Create a new interface that extends ISellable, named **IDiscountable**. Add a read-only property for the discount percentage. Also add a method that takes no parameters and returns void, named **ReduceSellingPrice**.

```
public interface IDiscountable : ISellable
{
   float DiscountPctg { get; }
   void ReduceSellingPrice();
}
```

5. Create a Book class that implements both IShelfable and ISellable.

```
public class Book : IShelfable, ISellable
{
```

6. Add three members to the Book class: name, price, and classification. Use a float for the price and a string for the others.

```
private string name;
private float price;
private string classification;
```

7. Write the constructor for the Book class.

```
public Book(
   string name,
   float price,
   string classification)
{
   this.name = name;
   this.price = price;
   this.classification = classification;
}
```

8. Implement the Name property (read-only):

```
public string Name
{
   get { return name; }
}
```

9. Implement the method PutOnShelf. For the purposes of this lab, just display the name of the book, its selling price, and its classification.

NOTE To display the selling price as a currency, use the C formatter. To display the second parameter as a price rather than using the substitution parameter {1}, you may instead use {1:C} (note the colon between the 1 and the C).

```
public void PutOnShelf ()
{
   Console.WriteLine(
      "Displaying {0} selling for {1:C} Classify as: {2}",
      name, price,classification);
}
```

10. Implement the Price property.

```
public float Price
{
   get { return price; }
   set { price = value; }
}
```

11. Implement the read-only Classification property.

```
public string Classification
{
   get { return classification; }
}
```

12. Implement a class CD to represent a music CD that implements both IShelfable and Idiscountable.

```
public class CD : IShelfable, IDiscountable
{
```

13. Create the four private fields for the CD class: cdName, price, discountPctg, and classification. The member's price and discountPctg will be floats, the other two will be strings. Initialize the classification to "Popular Music."

```
private string cdName;
private float price;
private float discountPctg;
private string classification = "Popular Music";
```

14. Implement the read-only property DiscountPctg.

```
public float DiscountPctg
{
   get { return discountPctg; }
}
```

15. Implement the Price property.

```
public float Price
{
   get { return price; }
   set { price = value; }
}
```

16. Implement the read-only property Classification.

```
public string Classification
{
   get { return classification; }
}
```

17. Implement the constructor.

```
public CD(string name, float price, float discount)
{
   this.cdName = name;
   this.price = price;
|   this.discountPctg = discount;
}
```

18. Implement the method PutOnShelf. For the purposes of this lab, just display the name of the book, its selling price, and its classification.

NOTE To display the selling price as a currency, use the C formatter. To display the second parameter as a price rather than using the substitution parameter {1} you may instead use {1:C} (note the colon between the 1 and the C).

```
public void PutOnShelf ()
{
   Console.WriteLine("Displaying {0} selling for {1:C}",
      cdName,price);
}
```

19. Implement ReduceSellingPrice by applying the available discount to the current price:

```
public void ReduceSellingPrice()
{
   price -= price * discountPctg;
}
```

20. In the Run method, create a CD object and put it on the shelf.

```
CD takeFive = new CD("Take Five", 15.75f, .10f);
takeFive.PutOnShelf();
```

21. Reduce the selling price of the CD and put it back on the shelf.

```
takeFive.ReduceSellingPrice();
takeFive.PutOnShelf();
```

22. Create a new book object and put it on the shelf.

```
Book progCS = new Book(
  "Programming C#",39.95f,"Programming");
progCS.PutOnShelf();
```

23. Set its price and reshelf it.

```
progCS.Price = 45.50f;
progCS.PutOnShelf();
```

24. Compile and run the program. It should look like Figure 13.

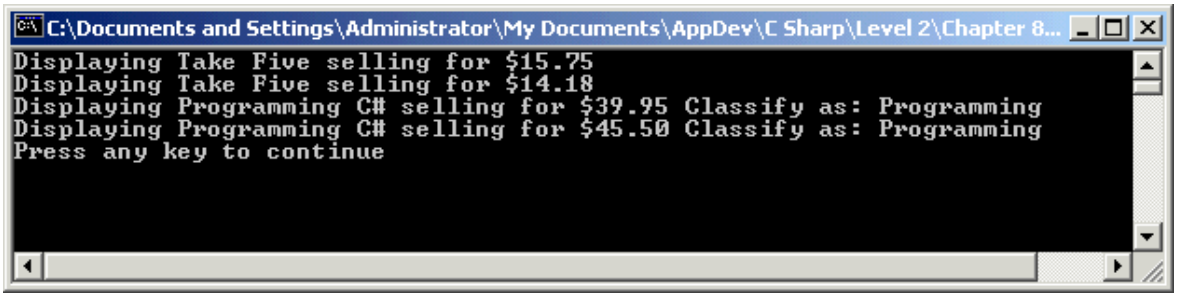

Figure 13. Final exercise results.