**Introducing Interfaces**

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.

Note the important distinction when using an Interface.  A class "implements" and interface as opposed to "inheriting" a base class.

**Creating Interfaces**

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.

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.  
}

*Note: Programming convention dictates that all interface names should begin with an "I".*

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.

Let's see a concrete example. Suppose that you want to develop a loyalty card scheme for an application related to a coffee company. 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();  
}

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.

**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.  
}

In most cases, whether you implement an interface implicitly or explicitly is an aesthetic choice. It does not make a difference in how your class compiles. 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

**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 an 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  
{  
}