Chapter 17

Interfaces

Interfaces

Objectives

After completing this unit you will be able to:

- **Use interfaces to implement polymorphic classes**
- **Use interfaces to implement a limited form of multiple inheritance**
- **Define your own interfaces**

Introduction

- **In C# *interface* is a keyword and has a very precise meaning.**
- **An interface is a reference type, similar to an abstract class, that *specifies* behavior.**
- **An interface can be thought of as a contract.**
- **A class or struct can "implement" an interface, and must adhere to the contract.**
- **Interfaces are a useful way to partition functionality.**
- **While a class in C# can inherit from only one other class, it can implement multiple interfaces.**
- **C# provides convenient facilities to query a class at runtime to see whether it supports a particular interface.**
- **We will see that interfaces in C# and .NET are conceptually very similar to interfaces in Microsoft's Component Object Model, but are *much* easier to work with.**

Interface Fundamentals

- **Object-oriented programming is a powerful paradigm for helping to design and implement large systems.**
  - Using classes helps us to achieve abstraction and encapsulation.
  - Classes are a natural decomposition of a large system into manageable parts.
  - Inheritance adds another tool for structuring our system, enabling us to factor out common parts into base classes, helping us to accomplish greater code reuse.
  - Interfaces provide yet another weapon for our arsenal.
- **The main purpose of an interface is to specify a contract independently of implementation.**
  - An interface has associated methods.
  - Each method has a signature, which specifies the parameters with their data types, and the data type of the return value.

Interfaces in C#

- **In C# *interface* is a keyword, and you define an interface in a manner similar to defining a class. Like classes, interfaces are reference types.**
- **The big difference is that there is no implementation code in an interface; it is pure specification.**
  - Also note that an interface can have properties as well as methods (it could also have other members, such as indexers).
  - The **IAccount** interface has read-only properties **Balance** and **Id** and the read-write property **Owner**.
  - Note the syntax, with a semicolon where the body of a method would be in a class.
  - There is also a semicolon after **set** and **get** in a property.
  - As a naming convention, interface names normally begin with a capital I.

```
interface IAccount
{
   void Deposit(decimal amount);
   void Withdraw(decimal amount);
   decimal Balance {get;}
   string Owner {get; set;}
   int Id {get;}
}
```

Interface Inheritance

- **Interfaces can inherit from other interfaces. Unlike classes in C#, for which there is only single inheritance, there can be multiple inheritance of interfaces.**
  - For example, the interface **IAccount** could be declared by inheriting from the two smaller interfaces, **IBasicAccount** and **IAccountInfo**.
  - When declaring a new interface in this way, you can also introduce additional methods, as illustrated for **IAccount2**.

```
interface IBasicAccount
{
   void Deposit(decimal amount);
   void Withdraw(decimal amount);
   decimal Balance {get;}
}

interface IAccountInfo
{
   string Owner {get; set;}
   int Id {get;}
}

interface IAccount : IBasicAccount, IAccountInfo
{
}

interface IAccount2 : IAccount
{
   void NewMethod();
}
```

Programming With Interfaces

- **It is very easy to program with interfaces in C#. You implement interfaces through classes, and you can cast a class reference to obtain an interface reference.**
- **You can call an interface method through either a class reference or an interface reference.**

Implementing Interfaces

- **In C# you specify that a class implements one or more interfaces by using the colon notation that is employed for class inheritance.**
  - A class can also inherit both from a class and from one or more interfaces.
  - In this case the base class should appear first in the derivation list after the colon.

```
class CheckingAccount : Account, IAccount,
                        IChecking
{
...
```

  - In this example the class **CheckingAccount** inherits from the class **Account**, and it implements the interfaces **IAccount** and **IChecking**.
  - The methods of the interfaces must all be implemented by **CheckingAccount**, either directly or by way of inheritance.
- **We will examine a full-blown example of interfaces with the account inheritance hierarchy later in the chapter, when we implement Step 6 of the case study.**
- **As a small example, consider the program *SmallInterface*.**
  - The class **Account** implements the interface **IBasicAccount**.

Implementing Interfaces (Cont'd)

```
// Account.cs

interface IBasicAccount
{
   void Deposit(decimal amount);
   void Withdraw(decimal amount);
   decimal Balance {get;}
}

public class Account : IBasicAccount
{
   private decimal balance;
   public Account(decimal balance)
   {
      this.balance = balance;
   }
   public void Deposit(decimal amount)
   {
      balance += amount;
   }
   public void Withdraw(decimal amount)
   {
      balance -= amount;
   }
   public decimal Balance
   {
      get
      {
         return balance;
      }
   }
}
```

Using an Interface

- **If you know your class supports an interface, you may simply call methods through a reference to a class instance.**
- **If you don't know whether your class implements the interface, you may try casting the class reference to the interface reference.**
  - If the class does not support the interface, you will get an **InvalidCastException**.

```
try
{
  IBasicAccount ifc2 = (IBasicAccount) acc2;
  ifc2.Deposit(25);
  Console.WriteLine("balance = {0}", ifc2.Balance);
}
catch (InvalidCastException e)
{
  Console.WriteLine("IBasicAccount is not supported");
  Console.WriteLine(e.Message);
}
```

  - In our example, we have two classes.
  - **Account** supports the interface **IBasicAccount**, and the other class **NoAccount** does not support the interface.
  - Both classes have the same set of methods and properties.

Demo: SmallInterface

- **We first work with the class *Account*, which does support the interface *IBasicAccount*.**
  - We are successful in calling the methods both through a class reference and through an interface reference.
- **Next we work with the class *NoAccount*.**
  - Although this class has the same methods as **Account**, in its declaration it does not indicate that it is implementing the interface **IBasicAccount**.
  - When we run the test program, we encounter an **InvalidCastException** when we attempt to cast the class reference to an interface reference.
- **Please examine and run the code online.**

Dynamic Use Of Interfaces

- **A powerful feature of interfaces is their use in dynamic scenarios, allowing us to write general code that can test whether an interface is supported by a class.**
- **If the interface is supported, our code can take advantage of it; otherwise our program can ignore the interface.**
  - We could in fact implement such dynamic behavior through exception handling, as illustrated previously.
  - Although entirely feasible, this approach is not very elegant.
  - C# provides two operators, **is** and **as**, that facilitate working with interfaces at runtime.
- **We will illustrate with some code snippets from a more full-blown account example, which is a continuation of our bank account case.**

Demo: TryInterfaces

- **We have two kinds of accounts, checking and savings.**
  - These accounts are implemented, respectively, in classes which inherit from **Account**. The class **CheckingAccount** implements the interface **IChecking** in addition to **IAccount** and **IStatement**. (The interfaces are defined in the file **AccountDefinitions.cs**.)
  - Our test program creates two checking account instances and one savings account instance, having IDs of 0, 1, and 2.
  - The object references are stored in an array and can be selected by using an index, which coincides with the ID.
  - Our command loop is generic and does not know which kind of account it is working on.
- **This kind of dynamic behavior is an extension of polymorphism, which we discussed in Chapter 14.**
  - With polymorphism you can call methods, which will behave differently depending upon the type of the object making the call. But the only methods you can call in this way are ones defined in the base class.
  - Using the concept of interfaces, you can call other methods, depending on whether additional interfaces are supported.
  - Thus if the interface **IChecking** is supported, you can call the **Fee** property. If the method **ISavings** is supported, you can call the **Interest** and **Rate** properties.

is Operator

- **The processing of the "fee" command illustrates doing a cast and catching an exception.**
- **A neater solution is to test for the interface *before* you do the cast.**
  - For this purpose you may use the C# **is** operator.
  - The "interest" command illustrates this point.

```
// use C# "is" operator
if (acc is ISavings)
{
  isav = (ISavings) acc;
  Console.WriteLine("interest = {0:C}",isav.Interest);
}
else
  Console.WriteLine("ISavings in not supported");
```

- **The *is* operator is not the most efficient solution, as a check of the type is made twice:**
  - When the is operator is invoked
  - When the actual type conversion is performed

as Operator

- **When you use the *as* operator, you obtain an interface reference directly.**
  - The interface reference is **null** if the class does not support the interface.
  - The type is checked only once in this scenario.

```
// use C# "as" operator
isav = acc as ISavings;
if (isav != null)
{
   isav = (ISavings) acc;
   Console.WriteLine("rate = {0}", isav.Rate);
}
else
   Console.WriteLine("ISavings in not supported");
```

  - If you are experienced with COM (Component Object Model), the operation of finding out if an interface is supported should be very familiar to you.

Bank Case Study: Step 6

- **We will now apply our knowledge of interfaces to do a little restructuring of the bank case study.**
- **Actually, comparatively little rewriting is needed, as C# interfaces integrate quite seamlessly into traditional class inheritance.**
  - One of the big benefits of using interfaces is that they raise the level of abstraction somewhat, helping you to understand the system by way of the interface contacts, without worrying about how the system is implemented.
  - It turns out that the implementation we have already constructed, using an abstract base class and two concrete derived classes, works perfectly for the interfaces.
  - But we now have a nice abstract contract, which could be implemented many different ways.
- **As usual, our case study code is in the *CaseStudy* directory for this chapter.**

Common Interfaces in Case Study – IAccount

- **We begin by examining the functionality of our base class *Account*.**
- **The methods and properties divide fairly naturally into two groups.**
  - The first group is concerned with operations on an **account** object (deposit or withdraw) and getting and setting fields.
  - This group of methods and properties constitutes the **IAccount** interface. These interfaces are defined in **AccountDefinitions.cs**.

```
interface IAccount
{
   void Deposit(decimal amount);
   void Withdraw(decimal amount);
   decimal Balance {get;}
   string Owner {get; set;}
   int Id {get;}
}
```

Apparent Redundancy

- **You may wonder why the *Balance*, *Owner*, and *Id* properties are in the *IAccount* interface when they are already in the abstract base class *Account*.**
- **Is there a redundancy here?**
  - Their functions in the interface and in the abstract base class are totally different.
  - In the interface, they are part of the contract.
  - Every class which implements the **IAccount** interface must support these properties. The abstract base class implements these properties.
- **You may also wonder about properties in the *IAccount* interface.**
- **Does this mean that somehow an interface can contain data?**
  - Not at all.
  - The properties specify behavior – how you can read and write a property value using a convenient notation.
  - The implementation of a property, where the data is stored, is in a class that implements the interface.

IStatement

- **The second group is concerned with getting a statement of an account and constitutes the *IStatement* interface.**

```
interface IStatement
{
   string FormatBalance();
   string GetStatement();
   int Transactions {get;}
   void Post();
   void MonthEnd();
   string Prompt {get;}
}
```

IStatement Methods

- ***FormatBalance*** **returns a string representation of the balance in proper currency format.**
- ***GetStatement*** **returns a string giving a complete statement for the account, including items such as the owner, id, balance, and number of transactions. Specific kinds of accounts will append supplementary information.**
- ***Transactions*** **returns a count of the number of transactions so far in the current month.**
- ***Post*** **will apply credits or debits for the current month. A checking account may have a debit of a fee, and a savings account a credit of interest.**
- ***MonthEnd*** **will initialize the account for the next month. The transactions count will be set to 0. For a savings account, the minimum balance (used for calculating interest) will be set to the current balance.**
- ***Prompt*** **returns a string indicating the type of account. The prompt string can be an aid to a user interface, giving a cue to the user about which kind of account is being worked on.**

IChecking

- **The *IChecking* interface is an additional interface supported only by checking accounts.**

```
interface IChecking
{
   decimal Fee {get;}
}
```

  - There is a single property, **Fee**, which is computed as the monthly fee owed for this account. (Note that the fee will not actually be debited from the balance until the **Post** method is invoked.)

ISavings

- **The ISavings interface is an additional interface supported only by savings accounts.**

```
interface ISavings
{
   decimal Interest {get;}
   decimal Rate {get; set;}
}
```

  - There are two properties.
  - **Rate** is an annual interest rate.
  - **Interest** is computed by multiplying **Rate/12** by the minimum balance.

The Implementation

- **There are no changes required to the *Account* class.**
  - This class is abstract, so it is not required to implement any interfaces.
  - This class contains common code that does not have to be provided separately by derived classes.
- **The *CheckingAccount* class now derives from one class, *Account*, and three interfaces, the two common interfaces *IAccount* and *IStatement*, and the special interface *IChecking*.**
  - None of the implementation code has to change.
  - We simply used the interfaces to codify the existing methods.

```
// CheckingAccount.cs - Step 6

using System;

public class CheckingAccount : Account, IAccount,
                               IStatement, IChecking
{
...
```

SavingsAccount

- **The *SavingsAccount* class also derives from one class, *Account*, and three interfaces, the two common interfaces *IAccount* and *IStatement*, and the special interface *ISavings*.**
  - Again, none of the implementation code has to change.

```
// SavingsAccount.cs - Step 6

using System;

public class SavingsAccount : Account, IAccount,
                              IStatement, ISavings
{
...
```

The Client

- **The client code, which uses the account classes, is where things get interesting.**
- **We can now write very general, dynamic code, which can tailor itself to the particular type of account it is working on, based on which interfaces are supported.**

The Client (Cont'd)

- **To keep things simple, in Step 6 we do not include the *Bank* class, but instead provide a special class *TestInterfaces*, which can be used to test the methods and properties of the various interfaces.**
  - The operations are performed on a set of three hardcoded accounts: two checking and one savings.
  - The structure of this class is similar to the **TryInterfaces** class we demonstrated earlier in the chapter, but the goals of the two classes are different.
  - The **TryInterfaces** class was intended to demonstrate different ways of dynamically working with interfaces (catch exceptions, use the **is** operator, and use the **as** operator).
  - The class **TestInterfaces** is intended to exercise the account classes fairly thoroughly (though not exhaustively; we do not provide test for every single method).
  - We pick the **as** operator as the generally most useful operator for working with interfaces dynamically, and use it throughout.
- **Please examine this code online.**

Resolving Ambiguity

- **When working with interfaces, an ambiguity can arise if a class implements two interfaces and each has a method with same name and signature.**
- **As an example, consider the following versions of the interfaces *IAccount* and *IStatement*. Each interface contains the method *Show*.**

```
interface IAccount
{
   void Deposit(decimal amount);
   void Withdraw(decimal amount);
   decimal Balance {get;}
   void Show();
}

interface IStatement
{
   int Transactions {get;}
   void Show();
```

- **How can the class specify implementations of these methods?**
- **The answer is to use the interface name to qualify the method, as illustrated in the program *Ambiguous*.**
  - The **IAccount** version **IAccount.Show** will display only the balance, and **IStatement.Show** will display both the number of transactions and the balance.

Access Modifier

- **You will notice that in the definition of the class Account, the qualified methods *IAccount.Show* and *IStatement.Show* do not have an access modifier such as public.**
  - Such qualified methods cannot be accessed through a reference to a class instance.
  - They can only be accessed through an interface reference of the type explicitly shown in the method definition.
  - The test program attempted to use a class instance reference, but the program would not compile, so this attempt was commented out.
  - In the test program we use interface references to make the calls to **Show**.

Summary

- **The term *interface* is widely used in computer programming to describe how parts of a large system fit together.**
- **In C# *interface* is a keyword and has a very precise meaning. An interface is a reference type, similar to an abstract class, that *specifies* behavior.**
- **An interface can be thought of as a contract. A class or struct can "implement" an interface, and must adhere to the contract.**
- **Interfaces are a useful way to partition functionality.**
- **The methods of a class can be grouped into related interfaces.**
- **While a class in C# can inherit from only one other class, it can implement multiple interfaces.**
- **Another benefit of interfaces is that they facilitate very dynamic programs.**
- **C# provides the operators *is* and *as* that can be used to query a class at runtime to see whether it supports a particular interface.**
- **Finally we looked at how to resolve an ambiguity that can arise if two interfaces have the same method.**