Chapter 12
OOP: Polymorphism, Interfaces and Operator Overloading

Visual C# 2012 How to Program

OBJECTIVES

In this chapter you'll learn:

- How polymorphism enables you to "program in the general" and make systems extensible.
- To use overridden methods to effect polymorphism.
- To create abstract classes and methods.
- To determine an object's type at execution time.
- To create `sealed` methods and classes.
- To declare and implement interfaces.
- To overload operators to enable them to manipulate objects.

12.6 `sealed` Methods and Classes (Cont.)

- A class that is declared `sealed` *cannot* be a base class (i.e., a class cannot extend a `sealed` class).
- All methods in a `sealed` class are implicitly `sealed`.
- Class `string` is a `sealed` class. This class cannot be extended, so apps that use `string`s can rely on the functionality of `string` objects as specified in the Framework Class Library.

Common Programming Error 12.4

Attempting to declare a derived class of a `sealed` class is a compilation error.

12.7 Case Study: Creating and Using Interfaces

Standardized Interactions

- Interfaces define and standardize the ways in which people and systems can interact with one another.
- A C# interface describes a set of methods that can be called on an object—to tell it, for example, to perform some task or return some piece of information.
- An **interface declaration** begins with the keyword `interface` and can contain only abstract methods, abstract properties, abstract indexers (not covered in this book) and abstract events (event are discussed in Chapter 14).
- All interface members are implicitly declared both `public` and `abstract`.
- An interface can extend one or more other interfaces to create a more elaborate interface that other classes can implement.

Common Programming Error 12.5

It's a compilation error to declare an interface member `public` or `abstract` explicitly, because they're redundant in interface-member declarations. It's also a compilation error to specify any implementation details, such as concrete method declarations, in an interface.

12.7 Case Study: Creating and Using Interfaces (Cont.)

Implementing an Interface

- To use an interface, a class must specify that it **implements** the interface by listing the interface after the colon (`:`) in the class declaration.
- A concrete class implementing an interface must declare each member of the interface with the signature specified in the interface declaration.
- A class that implements an interface but does *not* implement all its members is an abstract class—it must be declared `abstract` and must contain an `abstract` declaration for each unimplemented member of the interface.

Common Programming Error 12.6

Failing to define or declare any member of an interface in a class that implements the interface results in a compilation error.

12.7 Case Study: Creating and Using Interfaces (Cont.)

Common Methods for Unrelated Classes

- An interface is typically used when unrelated classes need to share common methods so that they can be processed polymorphically
- You can create an interface that describes the desired functionality, then implement this interface in any classes requiring that functionality.

12.7 Case Study: Creating and Using Interfaces (Cont.)

Interfaces vs. Abstract Classes

- An interface often is used in place of an `abstract` class when there is no default implementation to inherit—that is, no fields and no default method implementations.
- Like `abstract` classes, interfaces are typically `public` types, so they are normally declared in files by themselves with the same name as the interface and the `.cs` file-name extension.

12.7.1 Developing an `IPayable` Hierarchy

- To build an app that can determine payments for employees and invoices alike, we first create an interface named `IPayable`.
- Interface `IPayable` contains method `GetPaymentAmount` that returns a `decimal` amount to be paid for an object of any class that implements the interface.

Good Programming Practice 12.1

By convention, the name of an interface begins with `I`. This helps distinguish interfaces from classes, improving code readability.

Good Programming Practice 12.2

When declaring a method in an interface, choose a name that describes the method's purpose in a general manner, because the method may be implemented by a broad range of unrelated classes.

12.7.1 Developing an `IPayable` Hierarchy (Cont.)

UML Diagram Containing an Interface

- The UML class diagram in Fig. 12.10 shows the interface and class hierarchy used in our accounts-payable app.
- The UML distinguishes an interface from a class by placing the word “interface” in guillemets (« and ») above the interface name.
- The UML expresses the relationship between a class and an interface through a **realization**.

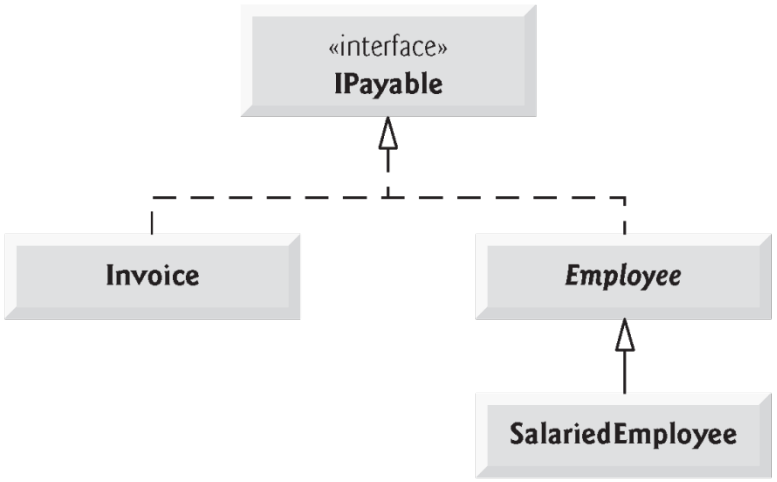

Fig. 12.10 | `IPayable` interface and class hierarchy UML class diagram.

12.7.2 Declaring Interface `IPayable`

- Interface `IPayable` is declared in Fig. 12.11.

```
1  // Fig. 12.11: IPayable.cs
2  // IPayable interface declaration.
3  public interface IPayable
4  {
5     decimal GetPaymentAmount(); // calculate payment; no implementation
6  } // end interface IPayable
```

Fig. 12.11 | `IPayable` interface declaration.

12.7.3 Creating Class `Invoice`

- We now create class `Invoice` (Fig. 12.12) represents a simple invoice that contains billing information for one kind of part.

```
1  // Fig. 12.12: Invoice.cs
2  // Invoice class implements IPayable.
3  using System;
4
5  public class Invoice : IPayable
6  {
7     private int quantity;
8     private decimal pricePerItem;
9
10    // property that gets and sets the part number on the invoice
11    public string PartNumber { get; set; }
12
13    // property that gets and sets the part description on the invoice
14    public string PartDescription { get; set; }
15
16    // four-parameter constructor
17    public Invoice( string part, string description, int count,
18       decimal price )
19    {
20       PartNumber = part;
21       PartDescription = description;
22       Quantity = count; // validate quantity via property
23       PricePerItem = price; // validate price per item via property
24    } // end four-parameter Invoice constructor
```

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 1 of 4.)

```
25
26      // property that gets and sets the quantity on the invoice
27      public int Quantity
28      {
29         get
30         {
31            return quantity;
32         } // end get
33         set
34         {
35            if ( value >= 0 ) // validate quantity
36               quantity = value;
37            else
38               throw new ArgumentOutOfRangeException( "Quantity",
39                  value, "Quantity must be >= 0" );
40         } // end set
41      } // end property Quantity
42
```

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 2 of 4.)

```
43      // property that gets and sets the price per item
44      public decimal PricePerItem
45      {
46         get
47         {
48            return pricePerItem;
49         } // end get
50         set
51         {
52            if ( value >= 0 ) // validate price
53               quantity = value;
54            else
55               throw new ArgumentOutOfRangeException( "PricePerItem",
56                  value, "PricePerItem must be >= 0" );
57         } // end set
58      } // end property PricePerItem
59
```

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 3 of 4.)

```
60      // return string representation of Invoice object
61      public override string ToString()
62      {
63         return string.Format(
64            "{0}: \n{1}: {2} ({3}) \n{4}: {5} \n{6}: {7:C}",
65            "invoice", "part number", PartNumber, PartDescription,
66            "quantity", Quantity, "price per item", PricePerItem );
67      } // end method ToString
68
69      // method required to carry out contract with interface IPayable
70      public decimal GetPaymentAmount()
71      {
72         return Quantity * PricePerItem; // calculate total cost
73      } // end method GetPaymentAmount
74   } // end class Invoice
```

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 4 of 4.)

12.7.3 Creating Class `Invoice` (Cont.)

- *C# does not allow derived classes to inherit from more than one base class, but it does allow a class to implement any number of interfaces.*
- To implement more than one interface, use a comma-separated list of interface names after the colon (`:`) in the class declaration.
- When a class inherits from a base class and implements one or more interfaces, the class declaration must list the base-class name before any interface names.

12.7.4 Modifying Class `Employee` to Implement Interface `IPayable`

- Figure 12.13 contains the `Employee` class, modified to implement interface `IPayable`.

```
1  // Fig. 12.13: Employee.cs
2  // Employee abstract base class.
3  public abstract class Employee : IPayable
4  {
5     // read-only property that gets employee's first name
6     public string FirstName { get; private set; }
7
8     // read-only property that gets employee's last name
9     public string LastName { get; private set; }
10
11    // read-only property that gets employee's social security number
12    public string SocialSecurityNumber { get; private set; }
13
14    // three-parameter constructor
15    public Employee( string first, string last, string ssn )
16    {
17       FirstName = first;
18       LastName = last;
19       SocialSecurityNumber = ssn;
20    } // end three-parameter Employee constructor
21
```

Fig. 12.13 | Employee abstract base class. (Part 1 of 2.)

```
22      // return string representation of Employee object
23      public override string ToString()
24      {
25         return string.Format( "{0} {1}\nsocial security number: {2}",
26            FirstName, LastName, SocialSecurityNumber );
27      } // end method ToString
28
29      // Note: We do not implement IPayable method GetPaymentAmount here, so
30      // this class must be declared abstract to avoid a compilation error.
31      public abstract decimal GetPaymentAmount();
32   } // end abstract class Employee
```

Fig. 12.13 | `Employee` abstract base class. (Part 2 of 2.)

12.7.5 Modifying Class `SalariedEmployee` for Use with `IPayable`

- Figure 12.14 contains a modified version of class `SalariedEmployee` that extends `Employee` and implements method `GetPaymentAmount`.
- The remaining `Employee` derived classes also must be modified to contain method `GetPaymentAmount` in place of `Earnings` to reflect the fact that `Employee` now implements `IPayable`.

```
1  // Fig. 12.14: SalariedEmployee.cs
2  // SalariedEmployee class that extends Employee.
3  using System;
4
5  public class SalariedEmployee : Employee
6  {
7     private decimal weeklySalary;
8
9     // four-parameter constructor
10    public SalariedEmployee( string first, string last, string ssn,
11       decimal salary ) : base( first, last, ssn )
12    {
13       WeeklySalary = salary; // validate salary via property
14    } // end four-parameter SalariedEmployee constructor
15
```

Fig. 12.14 | SalariedEmployee class that extends Employee. (Part 1 of 3.)

```
16      // property that gets and sets salaried employee's salary
17      public decimal WeeklySalary
18      {
19         get
20         {
21            return weeklySalary;
22         } // end get
23         set
24         {
25            if ( value >= 0 ) // validation
26               weeklySalary = value;
27            else
28               throw new ArgumentOutOfRangeException( "WeeklySalary",
29                  value, "WeeklySalary must be >= 0" );
30         } // end set
31      } // end property WeeklySalary
32
33      // calculate earnings; implement interface IPayable method
34      // that was abstract in base class Employee
35      public override decimal GetPaymentAmount()
36      {
37         return WeeklySalary;
38      } // end method GetPaymentAmount
```

Fig. 12.14 | SalariedEmployee class that extends Employee. (Part 2 of 3.)

```
39
40      // return string representation of SalariedEmployee object
41      public override string ToString()
42      {
43         return string.Format( "salaried employee: {0}\n{1}: {2:C}",
44            base.ToString(), "weekly salary", WeeklySalary );
45      } // end method ToString
46   } // end class SalariedEmployee
```

Fig. 12.14 | SalariedEmployee class that extends Employee. (Part 3 of 3.)

12.7.5 Modifying Class `SalariedEmployee` for Use with `IPayable`

- When a class implements an interface, the same *is-a* relationship provided by inheritance applies.

Software Engineering Observation 12.4

Inheritance and interfaces are similar in their implementation of the is-a relationship. An object of a class that implements an interface may be thought of as an object of that interface type. An object of any derived classes of a class that implements an interface also can be thought of as an object of the interface type.

Software Engineering Observation 12.5

The is-a relationship that exists between base classes and derived classes, and between interfaces and the classes that implement them, holds when passing an object to a method. When a method parameter receives an argument of a base class or interface type, the method polymorphically processes the object received as an argument.

12.7.6 Using Interface IPayable to Process Invoices and Employees Polymorphically

- `PayableInterfaceTest` (Fig. 12.15) illustrates that interface `IPayable` can be used to processes a set of `Invoice`s and `Employee`s polymorphically in a single app.

Software Engineering Observation 12.6

All methods of class `object` can be called by using a reference of an interface type—the reference refers to an object, and all objects inherit the methods of class `object`.

```
1  // Fig. 12.15: PayableInterfaceTest.cs
2  // Tests interface IPayable with disparate classes.
3  using System;
4
5  public class PayableInterfaceTest
6  {
7     public static void Main( string[] args )
8     {
9        // create four-element IPayable array
10       IPayable[] payableObjects = new IPayable[ 4 ];
11
12       // populate array with objects that implement IPayable
13       payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00M );
14       payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95M );
15       payableObjects[ 2 ] = new SalariedEmployee( "John", "Smith",
16          "111-11-1111", 800.00M );
17       payableObjects[ 3 ] = new SalariedEmployee( "Lisa", "Barnes",
18          "888-88-8888", 1200.00M );
19
20       Console.WriteLine(
21          "Invoices and Employees processed polymorphically:\n" );
22
```

Fig. 12.15 | Tests interface `IPayable` with disparate classes. (Part 1 of 3.)

```
23         // generically process each element in array payableObjects
24         foreach ( var currentPayable in payableObjects )
25         {
26            // output currentPayable and its appropriate payment amount
27            Console.WriteLine( "{0}\npayment due: {1:C}\n",
28               currentPayable, currentPayable.GetPaymentAmount() );
29         } // end foreach
30      } // end Main
31   } // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00
```

Fig. 12.15 | Tests interface `IPayable` with disparate classes. (Part 2 of 3.)

```
invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. 12.15 | Tests interface `IPayable` with disparate classes. (Part 3 of 3.)

12.7.7 Common Interfaces of the .NET Framework Class Library

- Figure 12.16 overviews several commonly used Framework Class Library interfaces.

Interface	Description
`IComparable`	As you learned in Chapter 3, C# contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare simple-type values. In Section 12.8 you'll see that these operators can be defined to compare two objects. Interface `IComparable` can also be used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, `CompareTo`, that compares the object that calls the method to the object passed as an argument to the method. Classes must implement `CompareTo` to return a value indicating whether the object on which it's invoked is less than (negative integer return value), equal to (0 return value) or greater than (positive integer return value) the object passed as an argument, using any criteria you specify. For example, if class `Employee` implements `IComparable`, its `CompareTo` method could compare `Employee` objects by their earnings amounts. Interface `IComparable` is commonly used for ordering objects in a collection such as an array. We use `IComparable` in Chapter 20, Generics, and Chapter 21, Collections.

Fig. 12.16 | Common interfaces of the .NET Framework Class Library. (Part 1 of 3.)

Interface	Description
`IComponent`	Implemented by any class that represents a component, including Graphical User Interface (GUI) controls (such as buttons or labels). Interface `IComponent` defines the behaviors that components must implement. We discuss `IComponent` and many GUI controls that implement this interface in Chapter 14, Graphical User Interfaces with Windows Forms: Part 1, and Chapter 15, Graphical User Interfaces with Windows Forms: Part 2.
`IDisposable`	Implemented by classes that must provide an explicit mechanism for *releasing* resources. Some resources can be used by only one program at a time. In addition, some resources, such as files on disk, are unmanaged resources that, unlike memory, cannot be released by the garbage collector. Classes that implement interface `IDisposable` provide a `Dispose` method that can be called to explicitly release resources. We discuss `IDisposable` briefly in Chapter 13, Exception Handling: A Deeper Look. You can learn more about this interface at `msdn.microsoft.com/en-us/library/system.idisposable.aspx`. The MSDN article *Implementing a Dispose Method* at `msdn.microsoft.com/en-us/library/fs2xkftw.aspx` discusses the proper implementation of this interface in your classes.

Fig. 12.16 | Common interfaces of the .NET Framework Class Library. (Part 2 of 3.)

Interface	Description
`IEnumerator`	Used for iterating through the elements *of a collection* (such as an array) one element at a time. Interface `IEnumerator` contains method `MoveNext` to move to the next element in a collection, method `Reset` to move to the position before the first element and property `Current` to return the object at the current location. We use `IEnumerator` in Chapter 21.

Fig. 12.16 | Common interfaces of the .NET Framework Class Library. (Part 3 of 3.)

12.8 Operator Overloading

- You can overload most operators to make them sensitive to the context in which they are used.

Class `ComplexNumber`

- Class `ComplexNumber` (Fig. 12.17) overloads the plus (+), minus (-) and multiplication (*) operators to enable programs to add, subtract and multiply instances of class `ComplexNumber` using common mathematical notation.

```
1  // Fig. 12.17: ComplexNumber.cs
2  // Class that overloads operators for adding, subtracting
3  // and multiplying complex numbers.
4  using System;
5
6  public class ComplexNumber
7  {
8     // read-only property that gets the real component
9     public double Real { get; private set; }
10
11    // read-only property that gets the imaginary component
12    public double Imaginary { get; private set; }
13
14    // constructor
15    public ComplexNumber( double a, double b )
16    {
17       Real = a;
18       Imaginary = b;
19    } // end constructor
20
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 1 of 3.)

```
21      // return string representation of ComplexNumber
22      public override string ToString()
23      {
24         return string.Format( "({0} {1} {2}i)",
25            Real, ( Imaginary < 0 ? "-" : "+" ), Math.Abs( Imaginary ) );
26      } // end method ToString
27
28      // overload the addition operator
29      public static ComplexNumber operator+ (
30         ComplexNumber x, ComplexNumber y )
31      {
32         return new ComplexNumber( x.Real + y.Real,
33            x.Imaginary + y.Imaginary );
34      } // end operator +
35
36      // overload the subtraction operator
37      public static ComplexNumber operator- (
38         ComplexNumber x, ComplexNumber y )
39      {
40         return new ComplexNumber( x.Real - y.Real,
41            x.Imaginary - y.Imaginary );
42      } // end operator -
43
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 2 of 3.)

```
44      // overload the multiplication operator
45      public static ComplexNumber operator* (
46         ComplexNumber x, ComplexNumber y )
47      {
48         return new ComplexNumber(
49            x.Real * y.Real - x.Imaginary * y.Imaginary,
50            x.Real * y.Imaginary + y.Real * x.Imaginary );
51      } // end operator *
52   } // end class ComplexNumber
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 3 of 3.)

12.8 Operator Overloading (Cont.)

- Keyword `operator`, followed by an operator symbol, indicates that a method overloads the specified operator.
- Methods that overload binary operators must take two arguments—the first argument is the *left* operand, and the second argument is the *right* operand.
- Overloaded operator methods must be `public` and `static`.

Software Engineering Observation 12.7

Overload operators to perform the same function or similar functions on class objects as the operators perform on objects of simple types. Avoid nonintuitive use of operators.

Software Engineering Observation 12.8

At least one parameter of an overloaded operator method must be a reference to an object of the class in which the operator is overloaded. This prevents you from changing how operators work on simple types.

12.8 Operator Overloading (Cont.)

Class `ComplexNumber`

- Class `ComplexTest` (Fig. 12.18) demonstrates the overloaded operators for adding, subtracting and multiplying `ComplexNumber`s.

```
1  // Fig. 12.18: ComplexTest.cs
2  // Overloading operators for complex numbers.
3  using System;
4
5  public class ComplexTest
6  {
7     public static void Main( string[] args )
8     {
9        // declare two variables to store complex numbers
10       // to be entered by user
11       ComplexNumber x, y;
12
13       // prompt the user to enter the first complex number
14       Console.Write( "Enter the real part of complex number x: " );
15       double realPart = Convert.ToDouble( Console.ReadLine() );
16       Console.Write(
17          "Enter the imaginary part of complex number x: " );
18       double imaginaryPart = Convert.ToDouble( Console.ReadLine() );
19       x = new ComplexNumber( realPart, imaginaryPart );
20
```

Fig. 12.18 | Overloading operators for complex numbers. (Part 1 of 3.)

```
21          // prompt the user to enter the second complex number
22          Console.Write( "\nEnter the real part of complex number y: " );
23          realPart = Convert.ToDouble( Console.ReadLine() );
24          Console.Write(
25             "Enter the imaginary part of complex number y: " );
26          imaginaryPart = Convert.ToDouble( Console.ReadLine() );
27          y = new ComplexNumber( realPart, imaginaryPart );
28
29          // display the results of calculations with x and y
30          Console.WriteLine();
31          Console.WriteLine( "{0} + {1} = {2}", x, y, x + y );
32          Console.WriteLine( "{0} - {1} = {2}", x, y, x - y );
33          Console.WriteLine( "{0} * {1} = {2}", x, y, x * y );
34       } // end method Main
35    } // end class ComplexTest
```

Fig. 12.18 | Overloading operators for complex numbers. (Part 2 of 3.)

```
Enter the real part of complex number x: 2
Enter the imaginary part of complex number x: 4

Enter the real part of complex number y: 4
Enter the imaginary part of complex number y: -2

(2 + 4i) + (4 - 2i) = (6 + 2i)
(2 + 4i) - (4 - 2i) = (-2 + 6i)
(2 + 4i) * (4 - 2i) = (16 + 12i)
```

Fig. 12.18 | Overloading operators for complex numbers. (Part 3 of 3.)