8a

Polymorphism, Interfaces & Operator Overloading, `struct` and `Nullable<T>`

OBJECTIVES

In this lecture you will learn:

- The concept of polymorphism and how it enables you to “program in the general.”
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
- Use `struct` and explicit interface qualification
- Use nullable types and the coalescing operator

OBJECTIVES

- How polymorphism makes systems extensible and maintainable.
- 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.

Outline

12.1 Introduction
12.2 Polymorphism Examples
12.3 Demonstrating Polymorphic Behavior
12.4 Abstract Classes and Methods
12.5 Case Study: Payroll System Using Polymorphism
12.6 `sealed` Methods and Classes
12.7 Case Study: Creating and Using Interfaces
12.8 Operator Overloading
12.9 Structs
12.10 Explicit Interface Member Name Qualification

12.1 Introduction

Polymorphism enables you to write applications that process objects that share the same base class in a class hierarchy as if they were all objects of the base class.

Polymorphism can improve extensibility.

12.2 Polymorphism Examples

If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` is a more specific version of a `Quadrilateral`.

Any operation that can be performed on a `Quadrilateral` object can also be performed on a `Rectangle` object.

These operations also can be performed on other `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.

The polymorphism occurs when an application invokes a method through a base-class variable.

The assignment of an instance of a derived class to a variable of the type of the base class is referred to as **upcasting**.

12.2 Polymorphism Examples (Cont.)

As another example, suppose we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`.

Each class inherits from the common base class `SpaceObject`, which contains method `Draw`.

A screen-manager application maintains a collection (e.g., a `SpaceObject` array) of references to objects of the various classes.

To refresh the screen, the screen manager periodically sends each object the same message—namely, `Draw`, while object responds in a unique way.

12.2 Polymorphism Examples (Cont.)

Software Engineering Observation 12.1

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. Only client code that instantiates new objects must be modified to accommodate new types.

12.3 Demonstrating Polymorphic Behavior

In a method call on an object, the type of the *actual referenced object*, not the type of the *reference*, determines which method is called.

An object of a derived class can be treated as an object of its base class.

A base-class object is not an object of any of its derived classes.

The *is-a* relationship applies from a derived class to its direct and indirect base classes, but not vice versa.

12.3 Demonstrating Polymorphic Behavior (Cont.)

The compiler allows the assignment of a base-class reference to a derived-class variable *if* we explicitly cast the base-class reference to the derived-class type

If an application needs to perform a derived-class-specific operation on a derived-class object referenced by a base-class variable, the base-class reference must be **downcasted** to a derived-class reference

- The example in Fig. 12.1 demonstrates three ways to use base-class and derived-class variables.

PolymorphismTest.cs

(1 of 3)

```
// Fig. 12.1: PolymorphismTest.cs
// Assigning base-class and derived-class references to base-class and
// derived-class variables.
using System;

public class PolymorphismTest
{
   public static void Main( string[] args )
   {
      // assign base-class reference to base-class variable
      CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
         "Sue", "Jones", "222-22-2222", 10000.00M, .06M );

      // assign derived-class reference to derived-class variable
      BasePlusCommissionEmployee4 basePlusCommissionEmployee =
         new BasePlusCommissionEmployee4( "Bob", "Lewis",
         "333-33-3333", 5000.00M, .04M, 300.00M );

      // invoke ToString and Earnings on base-class object
```

Create a new `CommissionEmployee3` object and assign its reference to a `CommissionEmployee3` variable.

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 1 of 3.)

```
      // using base-class variable
      Console.WriteLine( "{0} {1}:\n\n{2}\n{3}: {4:C}\n",
         "Call CommissionEmployee3's ToString with base-class reference",
         "to base class object", commissionEmployee.ToString(),
         "earnings", commissionEmployee.Earnings() );

      // invoke ToString and Earnings on derived-class object
      // using derived-class variable
      Console.WriteLine( "{0} {1}:\n\n{2}\n{3}: {4:C}\n",
         "Call BasePlusCommissionEmployee4's ToString with derived class",
         "reference to derived-class object",
         basePlusCommissionEmployee.ToString(),
         "earnings", basePlusCommissionEmployee.Earnings() );

      // invoke ToString and Earnings on derived-class object
      // using base-class variable
      CommissionEmployee3 commissionEmployee2 =
         basePlusCommissionEmployee;
      Console.WriteLine( "{0} {1}:\n\n{2}\n{3}: {4:C}",
         "Call BasePlusCommissionEmployee4's ToString with base class",
         "reference to derived-class object",
         commissionEmployee2.ToString(), "earnings",
         commissionEmployee2.Earnings() );
   } // end Main
} // end class PolymorphismTest
```

PolymorphismTest.cs

Use the reference `commissionEmployee` to invoke methods `ToString` and `Earnings`. Because `commissionEmployee` refers to a `CommissionEmployee3` object, base class `Commission-Employee3`'s version of the methods are called.

Assign the reference to derived-class object `basePlusCommissionEmployee` to a base-class `Commission-Employee3` variable.

Invoke methods `ToString` and `Earnings` on the base-class `CommisionEmployee3`, but the overriding derived-class's (`BasePlusCommissionEmployee4`'s) version of the methods are actually called.

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 2 of 3.)

```
Call CommissionEmployee3's ToString with base-class reference to base-class
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: $10,000.00
commission rate: 0.06
earnings: $600.00

Call BasePlusCommissionEmployee4's ToString with derived-class reference to
derived-class object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $300.00
earnings: $500.00

Call BasePlusCommissionEmployee4's ToString with base-class reference to derived-
class object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $300.00
earnings: $500.00
```

PolymorphismTest.cs

(3 of 3)

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 3 of 3.)

12.3 Demonstrating Polymorphic Behavior (Cont.)

When the compiler encounters a method call made through a variable, it determines if the method can be called by checking the *variable's* class type.

At execution time, *the type of the object to which the variable refers* determines the actual method to use.

12.4 Abstract Classes and Methods

Abstract classes, or **abstract base classes** cannot be used to instantiate objects.

Abstract base classes are too general to create real objects—they specify only what is common among derived classes.

Classes that can be used to instantiate objects are called **concrete classes**.

Concrete classes provide the specifics that make it reasonable to instantiate objects.

12.4 Abstract Classes and Methods (Cont.)

An abstract class normally contains one or more **abstract methods**, which have the keyword `abstract` in their declaration.

A class that contains abstract methods must be declared as an abstract class even if it contains concrete (nonabstract) methods.

Abstract methods do not provide implementations.

12.4 Abstract Classes and Methods (Cont.)

abstract property declarations have the form:

```
public abstract PropertyType MyProperty
{
 get;
 set;
} // end abstract property
```

An abstract property may omit implementations for the `get` accessor, the `set` accessor or both.

Concrete derived classes must provide implementations for *every* accessor declared in the abstract property.

12.4 Abstract Classes and Methods (Cont.)

Constructors and `static` methods **cannot** be declared `abstract`.

Software Engineering Observation 12.2

An abstract class declares common attributes and behaviors of the various classes that inherit from it, either directly or indirectly, in a class hierarchy. An abstract class typically contains one or more abstract methods or properties that concrete derived classes must override.

12.4 Abstract Classes and Methods (Cont.)

Working with `abstract` methods

`abstract` methods are **`virtual`** implicitly

`override` methods can override **`abstract`** methods in derived classes

`abstract` methods can override base class methods declared as **`virtual`**

`abstract` methods can override base class methods declared as **`override`**

12.4 Abstract Classes and Methods (Cont.)

Examples with `abstract` methods

```
class A {
        public virtual string  Method() {}
}
class C : A {
        public override string  Method() {}
   public virtual string  NewMethod() {}
}
abstract class B : C {
        public abstract override string Method();
   public abstract override string NewMethod();
}
```

12.4 Abstract Classes and Methods (Cont.)

```
class A
    {
        public virtual void M() { Console.WriteLine("A class\n"); }
    }
class B : A
    {
        new public void M(){Console.WriteLine("B class\n");}
    }
class C : B
    {
        new public virtual void M()
        {
            Console.WriteLine("C class\n");
        }
    }
```

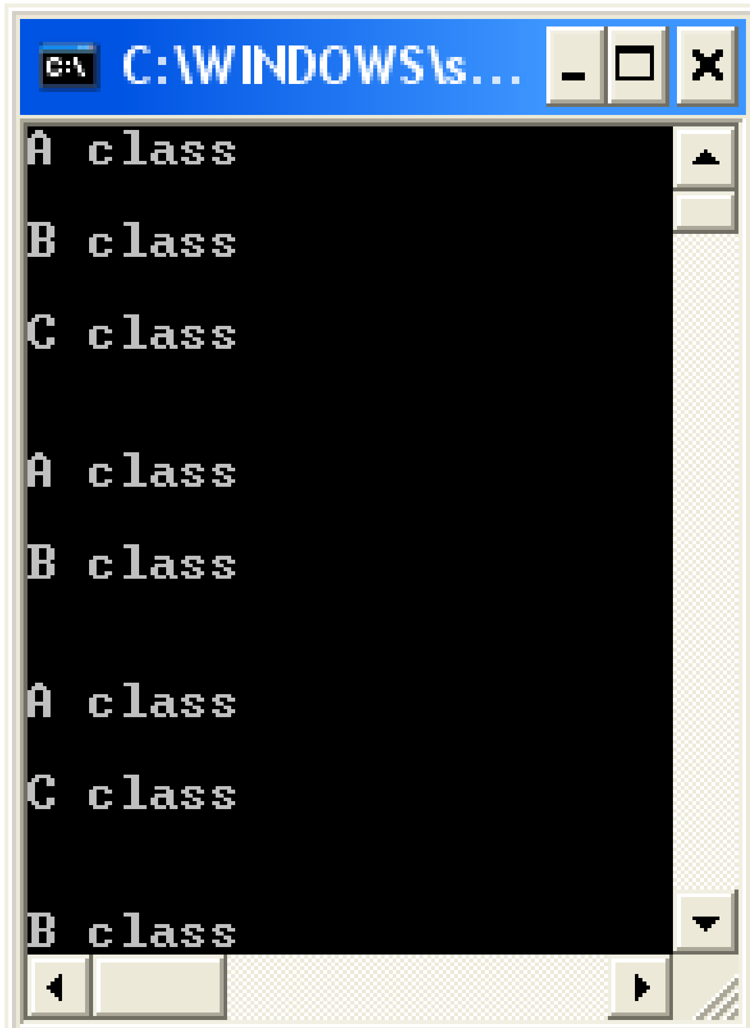
12.4 Abstract Classes and Methods (Cont.)

```
A a = new A();
B b = new B();
C c = new C();
a.M();
b.M();
c.M();

a = b;
a.M();
b.M();

a = c;
a.M();
c.M();

b = c;
b.M();
```



12.4 Abstract Classes and Methods (Cont.)

Common Programming Error 12.1

Attempting to instantiate an object of an abstract class is a compilation error.

Common Programming Error 12.2

Failure to implement a base class's abstract methods and properties in a derived class is a compilation error unless the derived class is also declared `abstract`.

12.4 Abstract Classes and Methods (Cont.)

We can use abstract base classes to declare variables that can hold references to objects of any concrete classes derived from those abstract classes.

You can use such variables to manipulate derived-class objects polymorphically and to invoke `static` methods declared in those abstract base classes.

It is common in object-oriented programming to declare an **iterator class** that can traverse all the objects in a collection.

12.5 Case Study: Payroll System Using Polymorphism

In this section, we create an enhanced employee hierarchy to solve the following problem:

A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales, and salaried-commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

We use `abstract` class `Employee` to represent the general concept of an employee.

`SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee` extend `Employee`.

Class `BasePlusCommissionEmployee`—which extends `CommissionEmployee`—represents the last employee type.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

The UML class diagram in Fig. 12.2 shows the inheritance hierarchy for our polymorphic employee payroll application.

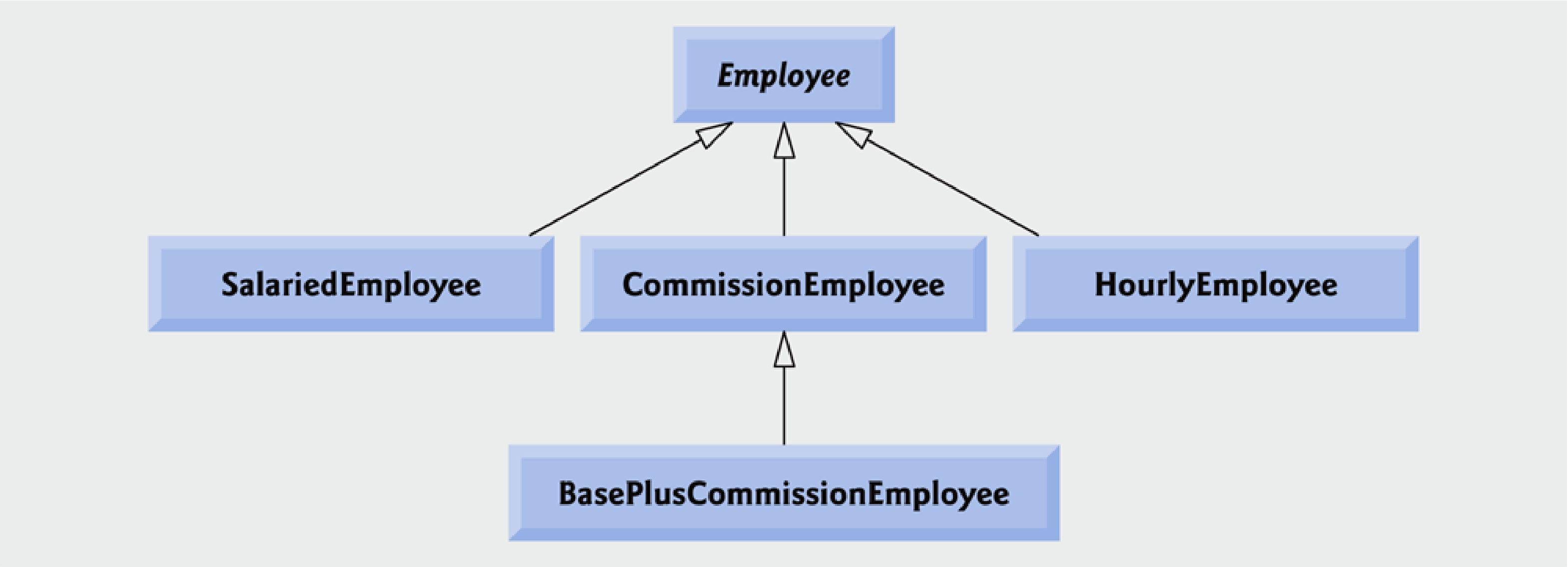


Fig. 12.2 | `Employee` hierarchy UML class diagram

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

Software Engineering Observation 12.3

A derived class can inherit "interface" or "implementation" from a base class. Hierarchies designed for implementation inheritance tend to have their functionality high in the hierarchy. Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

12.5.1 Creating Abstract Base Class `Employee`

Class `Employee` provides methods `Earnings` and `ToString`, in addition to the properties that manipulate `Employee`'s instance variables.

Each earnings calculation depends on the employee's class, so we declare `Earnings` as `abstract`.

The application iterates through the array and calls method `Earnings` for each `Employee` object. C# processes these method calls polymorphically.

Each derived class overrides method `ToString` to create a string representation of an object of that class.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

The diagram in Fig. 12.3 shows each of the five classes in the hierarchy down the left side and methods `Earnings` and `ToString` across the top.

	Earnings	ToString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage* hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate* base salary: *baseSalary*

Fig. 12.3 | Polymorphic interface for the `Employee` hierarchy classes.

- The `Employee` class's declaration is shown in Fig. 12.4.

Employee.cs

(1 of 2)

```
// Fig. 12.4: Employee.cs
// Employee abstract base class.
public abstract class Employee
{
   // read-only property that gets employee's first name
   public string FirstName { get; private set; }

   // read-only property that gets employee's last name
   public string LastName { get; private set; }

   // read-only property that gets employee's social security number
   public string SocialSecurityNumber { get; private set; }

   // three-parameter constructor
   public Employee( string first, string last, string ssn )
   {
      FirstName = first;
      LastName = last;
      SocialSecurityNumber = ssn;
   } // end three-parameter Employee constructor
```

Fig. 12.4 | `Employee` abstract base class. (Part 1 of 2.)

Employee.cs

(1 of 2)

```

   // return string representation of Employee object, using properties
   public override string ToString()
   {
      return string.Format( "{0} {1}\nsocial security number: {2}",
         FirstName, LastName, SocialSecurityNumber );
   } // end method ToString

   // abstract method overridden by derived classes
   public abstract decimal Earnings(); // no implementation here
} // end abstract class Employee
```

The `Employee` class includes an `abstract` method `Earnings`, which must be implemented by concrete derived classes.

Fig. 12.4 | `Employee` abstract base class. (Part 2 of 2.)

SalariedEmployee.cs

(1 of 2)

```
// Fig. 12.5: SalariedEmployee.cs
// SalariedEmployee class that extends Employee.
public class SalariedEmployee : Employee
{
   private decimal weeklySalary;

   // four-parameter constructor
   public SalariedEmployee( string first, string last, string ssn,
      decimal salary ) : base( first, last, ssn )
   {
      WeeklySalary = salary; // validate salary via property
   } // end four-parameter SalariedEmployee constructor

   // property that gets and sets salaried employee's salary
   public decimal WeeklySalary
   {
      get
      {
         return weeklySalary;
      } // end get
```

`SalariedEmployee` extends `Employee`.

Using the base class constructor to initialize the `private` variables not inherited from the base class.

Fig. 12.5 | `SalariedEmployee` class that extends `Employee`. (Part 1 of 2.)

SalariedEmployee.cs


```
      set
      {
         weeklySalary = ( ( value >= 0 ) ? value : 0 ); // validation
      } // end set
   } // end property WeeklySalary

   // calculate earnings; override abstract method Earnings in Employee
   public override decimal Earnings()
   {
      return WeeklySalary;
   } // end method Earnings

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

Method `Earnings` overrides `Employee`'s abstract method `Earnings` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary.

Method `ToString` overrides `Employee` method `ToString`.

Fig. 12.5 | `SalariedEmployee` class that extends `Employee`. (Part 2 of 2.)

- Class `HourlyEmployee` (Fig. 12.6) also extends class `Employee`.

```
// Fig. 12.6: HourlyEmployee.cs
// HourlyEmployee class that extends Employee.
public class HourlyEmployee : Employee
{
   private decimal wage; // wage per hour
   private decimal hours; // hours worked for the week

   // five-parameter constructor
   public HourlyEmployee( string first, string last, string ssn,
      decimal hourlyWage, decimal hoursWorked )
      : base( first, last, ssn )
   {
      Wage = hourlyWage; // validate hourly wage via property
      Hours = hoursWorked; // validate hours worked via property
   } // end five-parameter HourlyEmployee constructor

   // property that gets and sets hourly employee's wage
   public decimal Wage
   {
      get
      {
         return wage;
      } // end get
```

HourlyEmployee.cs

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 1 of 3.)

HourlyEmployee.cs

(2 of 3)

```
      set
      {
         wage = ( value >= 0 ) ? value : 0; // validation
      } // end set
   } // end property Wage

   // property that gets and sets hourly employee's hours
   public decimal Hours
   {
      get
      {
         return hours;
      } // end get
      set
      {
         hours = ( ( value >= 0 ) && ( value <= 168 ) ) ?
                 value : 0; // validation
      } // end set
   } // end property Hours
```

Method `ToString` overrides `Employee` method `ToString`.

The `set` accessor in property `Hours` ensures that `hours` is in the range `0`–`168` (the number of hours in a week).

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 2 of 3.)

HourlyEmployee.cs

(3 of 3)

```

    // calculate earnings; override Employee’s abstract method Earnings
    public override decimal Earnings()
    {
       if ( Hours <= 40 ) // no overtime
          return Wage * Hours;
       else
          return ( 40 * Wage ) + ( ( Hours - 40 ) * Wage * 1.5M );
    } // end method Earnings

    // return string representation of HourlyEmployee object
    public override string ToString()
    {
       return string.Format(
          "hourly employee: {0}\n{1}: {2:C}; {3}: {4:F2}",
          base.ToString(), "hourly wage", Wage, "hours worked", Hours );
    } // end method ToString
 } // end class HourlyEmployee
```

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 3 of 3.)

- Class `CommissionEmployee` (Fig. 12.7) extends class `Employee`.

`CommissionEmployee.cs`

(1 of 3)

```
// Fig. 12.7: CommissionEmployee.cs
// CommissionEmployee class that extends Employee.
public class CommissionEmployee : Employee
{
   private decimal grossSales; // gross weekly sales
   private decimal commissionRate; // commission percentage

   // five-parameter constructor
   public CommissionEmployee( string first, string last, string ssn,
      decimal sales, decimal rate ) : base( first, last, ssn )
   {
      GrossSales = sales; // validate gross sales via property
      CommissionRate = rate; // validate commission rate via property
   } // end five-parameter CommissionEmployee constructor

   // property that gets and sets commission employee's commission rate
   public decimal CommissionRate
   {
      get
      {
         return commissionRate;
      } // end get
```

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 1 of 3.)

CommissionEmployee.cs

(2 of 3)

```
      set
      {
         commissionRate = ( value > 0 && value < 1 ) ?
                              value : 0; // validation
      } // end set
   } // end property CommissionRate

   // property that gets and sets commission employee's gross sales
   public decimal GrossSales
   {
      get
      {
         return grossSales;
      } // end get
      set
      {
         grossSales = ( value >= 0 ) ? value : 0; // validation
      } // end set
   } // end property GrossSales
```

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 2 of 3.)

CommissionEmployee.cs

(3 of 3)

```

   // calculate earnings; override abstract method Earnings in Employee
   public override decimal Earnings()
   {
      return CommissionRate * GrossSales;
   } // end method Earnings

   // return string representation of CommissionEmployee object
   public override string ToString()
   {
      return string.Format( "{0}: {1}\n{2}: {3:C}\n{4}: {5:F2}",
         "commission employee", base.ToString(),
         "gross sales", GrossSales, "commission rate", CommissionRate );
   } // end method ToString
} // end class CommissionEmployee
```

Calling base-class method `ToString` to obtain the `Employee`-specific information.

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 3 of 3.)

- Class `BasePlusCommissionEmployee` (Fig. 12.8) extends class `CommissionEmployee` and therefore is an indirect derived class of class `Employee`.


```
// Fig. 12.8: BasePlusCommissionEmployee.cs
// BasePlusCommissionEmployee class that extends CommissionEmployee.
public class BasePlusCommissionEmployee : CommissionEmployee
{
   private decimal baseSalary; // base salary per week

   // six-parameter constructor
   public BasePlusCommissionEmployee( string first, string last,
      string ssn, decimal sales, decimal rate, decimal salary )
      : base( first, last, ssn, sales, rate )
   {
      BaseSalary = salary; // validate base salary via property
   } // end six-parameter BasePlusCommissionEmployee constructor

   // property that gets and sets
   // base-salaried commission employee's base salary
   public decimal BaseSalary
   {
      get
      {
         return baseSalary;
      } // end get
```

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 1 of 2.)

BasePlusCommissionEmployee.cs

(2 of 2)

```
    set
    {
      baseSalary = ( value >= 0 ) ? value : 0; // validation
    } // end set
  } // end property BaseSalary

  // calculate earnings; override method Earnings in CommissionEmployee
  public override decimal Earnings()
  {
    return BaseSalary + base.Earnings();
  } // end method Earnings

  // return string representation of BasePlusCommissionEmployee object
  public override string ToString()
  {
    return string.Format( "base-salaried {0}; base salary: {1:C}",
      base.ToString(), BaseSalary );
  } // end method ToString
} // end class BasePlusCommissionEmployee
```

Method `Earnings` calls the base class's `Earnings` method to calculate the commission-based portion of the employee's earnings.

`BasePlusCommissionEmployee`'s `ToString` method creates a string that contains `"base-salaried"`, followed by the `string` obtained by invoking base class `CommissionEmployee`'s `ToString` method (a good example of code reuse) then the base salary.

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 2 of 2.)

- The application in Fig. 12.9 tests our `Employee` hierarchy.


```
// Fig. 12.9: PayrollSystemTest.cs
// Employee hierarchy test application.
using System;

public class PayrollSystemTest
{
   public static void Main( string[] args )
   {
      // create derived-class objects
      SalariedEmployee salariedEmployee =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00M );
      HourlyEmployee hourlyEmployee =
         new HourlyEmployee( "Karen", "Price",
         "222-22-2222", 16.75M, 40.0M );
      CommissionEmployee commissionEmployee =
         new CommissionEmployee( "Sue", "Jones",
         "333-33-3333", 10000.00M, .06M );
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee( "Bob", "Lewis",
         "444-44-4444", 5000.00M, .04M, 300.00M );

```

Create objects of each of the four concrete `Employee` derived classes.

Fig. 12.9 | `Employee` hierarchy test application. (Part 1 of 6.)

PayrollSystemTest.cs

(2 of 6)

```
        Console.WriteLine( "Employees processed individually:\n" );

        Console.WriteLine( "{0}\nearned: {1:C}\n",
           salariedEmployee, salariedEmployee.Earnings() );
        Console.WriteLine( "{0}\nearned: {1:C}\n",
           hourlyEmployee, hourlyEmployee.Earnings() );
        Console.WriteLine( "{0}\nearned: {1:C}\n",
           commissionEmployee, commissionEmployee.Earnings() );
        Console.WriteLine( "{0}\nearned: {1:C}\n",
           basePlusCommissionEmployee,
           basePlusCommissionEmployee.Earnings() );

        // create four-element Employee array
        Employee[] employees = new Employee[ 4 ];

        // initialize array with Employees of derived types
        employees[ 0 ] = salariedEmployee;
        employees[ 1 ] = hourlyEmployee;
        employees[ 2 ] = commissionEmployee;
        employees[ 3 ] = basePlusCommissionEmployee;
```

Each object's `ToString` method is called implicitly.

Fig. 12.9 | `Employee` hierarchy test application. (Part 2 of 6.)

```
      Console.WriteLine( "Employees processed polymorphically:\n" );

      // generically process each element in array employees
      foreach ( var currentEmployee in employees )
      {
         Console.WriteLine( currentEmployee ); // invokes ToString

         // determine whether element is a BasePlusCommissionEmployee
         if ( currentEmployee is BasePlusCommissionEmployee )
         {
            // downcast Employee reference to
            // BasePlusCommissionEmployee reference
            BasePlusCommissionEmployee employee =
               ( BasePlusCommissionEmployee ) currentEmployee;

            employee.BaseSalary *= 1.10M;
            Console.WriteLine(
               "new base salary with 10% increase is: {0:C}",
               employee.BaseSalary );
         } // end if
```

d calls are resolved at
ion time, based on the
f the object referenced by
iable.

s operator is used to
ine whether a particular
oyee object's type is
PlusCommissionEmp
e.

casting current-
oyee from type
oyee to type
PlusCommissionEmp
e.

Fig. 12.9 | Employee hierarchy test application. (Part 3 of 6.)

PayrollSystemTest.cs


```
            Console.WriteLine(
               "earned {0:C}\n", currentEmployee.Earnings() );
         } // end foreach

         // get type name of each object in employees array
         for ( int j = 0; j < employees.Length; j++ )
            Console.WriteLine( "Employee {0} is a {1}", j,
               employees[ j ].GetType() );
      } // end Main
   } // end class PayrollSystemTest
```

Method calls are resolved at execution time, based on the type of the object referenced by the variable.

Method `GetType` returns an object of class `Type`, which contains information about the object's type.

```
Employees processed individually:

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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00
```

(continued on next page…)

Fig. 12.9 | `Employee` hierarchy test application. (Part 4 of 6.)

(continued from previous page…)

```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00
commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00
commission rate: 0.04; base salary: $300.00
earned: $500.00


Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
```

```
weekly salary: $800.00
earned $800.00
```

(continued on previous page…)

PayrollSystemTest.cs

(5 of 6)

Fig. 12.9 | `Employee` hierarchy test application. (Part 5 of 6.)

(continued from previous page…)

```
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00
commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00
commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

PayrollSystemTest.cs

(6 of 6)

Fig. 12.9 | `Employee` hierarchy test application. (Part 6 of 6.)

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

Common Programming Error 12.3

Assigning a base-class variable to a derived-class variable (without an explicit downcast) is a compilation error.

Software Engineering Observation 12.4

If at execution time the reference to a derived-class object has been assigned to a variable of one of its direct or indirect base classes, it is acceptable to cast the reference stored in that base-class variable back to a reference of the derived-class type. Before performing such a cast, use the `is` operator to ensure that the object is indeed an object of an appropriate derived-class type.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

Common Programming Error 12.4

When downcasting an object, an `InvalidCastException` (of namespace System) occurs if at execution time the object does not have an is-a relationship with the type specified in the cast operator. An object can be cast only to its own type or to the type of one of its base classes.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- You can avoid a potential `InvalidCastException` by using the **as operator** to perform a **downcast** rather than a cast operator.
  - If the downcast is invalid, the expression will be `null` instead of throwing an exception.
- Method `GetType` returns an object of class Type (of namespace `System`), which contains information about the object's type, including its class name, the names of its methods, and the name of its base class.
- The `Type` class's `ToString` method returns the class name.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

Example using the keyword **as**

```
      Console.WriteLine( "Employees processed polymorphically:\n" );

      // generically process each element in array employees
      foreach ( var currentEmployee in employees )
      {
         Console.WriteLine( currentEmployee ); // invokes ToString
         BasePlusCommissionEmployee employee = // convert to BasePlusCommissionEmployee
                                   currentEmployee  as  BasePlusCommissionEmployee ;
         if (employee  != null )
         {
            /  Employee reference to
            // BasePlusCommissionEmployee object

            employee.BaseSalary *= 1.10M;
            Console.WriteLine(
               "new base salary with 10% increase is: {0:C}",
               employee.BaseSalary );
         } // end if
```

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

12.5.7 Summary of the Allowed Assignments Between Base-Class and Derived-Class Variables

- Assigning a base-class reference to a base-class variable is straightforward.
- Assigning a derived-class reference to a derived-class variable is straightforward.
- Assigning a derived-class reference to a base-class variable is safe, because the derived-class object *is an* object of its base class. However, this reference can be used to refer only to base-class members.
- Attempting to assign a base-class reference to a derived-class variable is a compilation error. To avoid this error, the base-class reference must be cast to a derived-class type explicitly.

12.6 `sealed` Methods and Classes

- A method declared **`sealed`** in a base class cannot be overridden in a derived class.
- Methods that are declared `private` are implicitly `sealed`.
- Methods that are declared `static` also are implicitly `sealed`, because `static` methods cannot be overridden either.
- A derived-class method declared both `override` and `sealed` can override a base-class method, but cannot be overridden in classes further down the inheritance hierarchy.
- Calls to `sealed` methods are resolved at compile time—this is known as **static binding**.

12.6 `sealed` Methods and Classes (Cont.)

Performance Tip 12.1

The compiler can decide to inline a `sealed` method call and will do so for small, simple `sealed` methods. Inlining does not violate encapsulation or information hiding, but does improve performance, because it eliminates the overhead of making a method call.

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 applications that use `string`s can rely on the functionality of `string` objects as specified in the Framework Class Library.

12.6 `sealed` Methods and Classes (Cont.)

Common Programming Error 12.5

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

Software Engineering Observation 12.5

In the Framework Class Library, the vast majority of classes are not declared sealed. This enables inheritance and polymorphism—the fundamental capabilities of object-oriented programming.

12.5 Case Study: Payroll System Using …

Working with Virtual methods

Purpose-

- **`virtual`** allows a method overriding in derived classes
- An **`override`** method specifies another implementation of a **`virtual`** method

Restrictions:

- Cannot declare **`virtual`** methods as **`static`** (polymorphism works on objects not on classes!)
- **`Virtual`** methods cannot be declared as **`private`** (prevents overriding in derived classes)
- Sequence of method definition allowing overriding

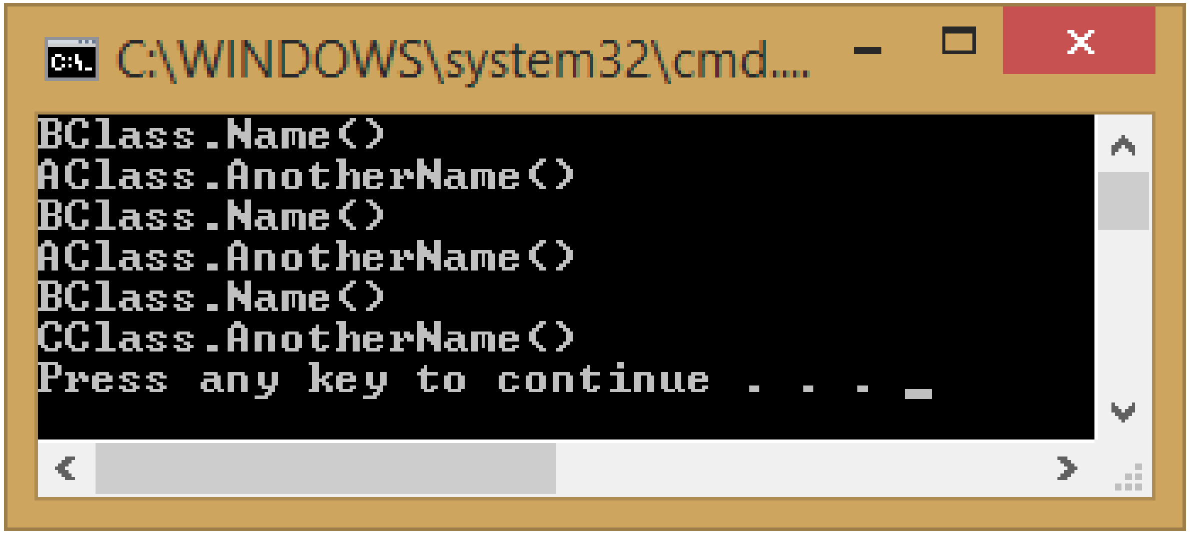
`(abstract, virtual or new virtual)`→

→ `override`→…→`override`

→ `(sealed or new)`

12.5 Case Study: Payroll System Using …

```
class Program
{
    static void Main(string[] args)
    {
        Aclass ac = new Aclass();
        Bclass bc = new Bclass();
        Cclass cc = new Cclass();
        ac = cc;
        ac.Name();
        ac.AnotherName();
        ac = bc;
        ac.Name();
        ac.AnotherName();
        bc = cc;
        bc.Name();
        bc.AnotherName();
    }
}
class Aclass
{
    public virtual void Name() { Console.WriteLine("{0}", "AClass.Name()"); }
    public void AnotherName() { Console.WriteLine("{0}", "AClass.AnotherName()"); }
}
class Bclass : Aclass
{
    public override void Name() { Console.WriteLine("{0}", "BClass.Name()"); }
    new public virtual void AnotherName() { Console.WriteLine("{0}", "BClass.AnotherName()"); }
}
class Cclass : Bclass
{
    new public void Name() { Console.WriteLine("{0}", "CClass.Name()"); }
    public override void AnotherName() { Console.WriteLine("{0}", "CClass.AnotherName()"); }
}// or public  sealed  override void AnotherName() {    }
```



12.7 Case Study: Creating and Using Interfaces

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, properties, indexers and events.

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.

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

Common Programming Error 12.6

It is a compilation error to declare an interface member public or abstract explicitly, because they are redundant in interface-member declarations. It is 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.)

Interface types, not being classes, are not derived from **`object`**. They are all *convertible* to object, to be sure, because we know that at runtime the instance will be a concrete type.

Consequently, a variable of **`interface`** type provides access at run- time to all the methods of class **`object`** in addition to the methods declared in the particular **`interface`**.

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

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 12.7

Failing to 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.)

An interface is typically used when disparate (i.e., unrelated) classes need to share common methods so that they can be processed polymorphically

A programmer can create an interface that describes the desired functionality, then implement this interface in any classes requiring that functionality.

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 Case Study: Creating and Using Interfaces (Cont.)

12.7.1 Developing an `IPayable` Hierarchy

- To build an application 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.

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

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 Case Study: Creating and Using Interfaces (Cont.)

The UML class diagram in Fig. 12.10 shows the interface and class hierarchy used in our accounts-payable application.

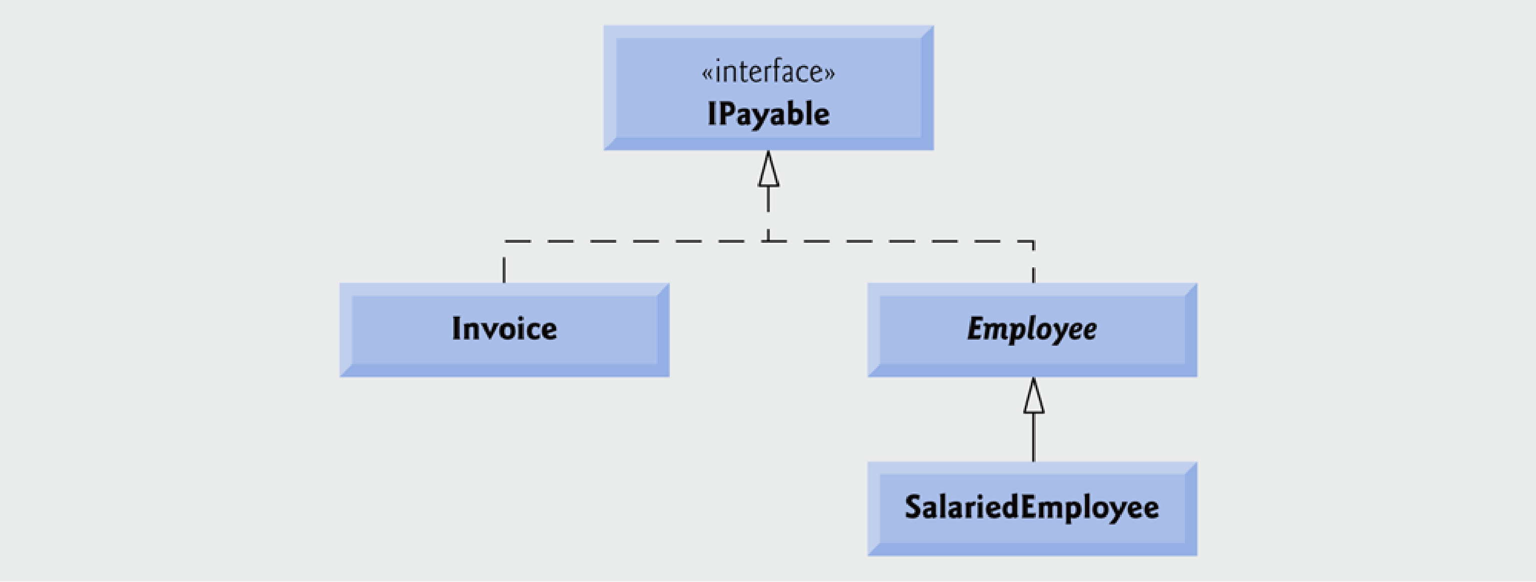


Fig. 12.10 | `IPayable` interface and class hierarchy UML class diagram.

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

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**.

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

IPayable.cs

```
// Fig. 12.11: IPayable.cs
// IPayable interface declaration.
public interface IPayable
{
   decimal GetPaymentAmount(); // calculate payment; no implementation
} // end interface IPayable
```

Fig. 12.11 | `IPayable` interface declaration.

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

```
// Fig. 12.12: Invoice.cs
// Invoice class implements IPayable.
public class Invoice : IPayable
{
   private int quantity;
   private decimal pricePerItem;

   // property that gets and sets the part number on the invoice
   public string PartNumber { get; set; }

   // property that gets and sets the part description on the invoice
   public string PartDescription { get; set; }

   // four-parameter constructor
   public Invoice( string part, string description, int count,
      decimal price )
   {
      PartNumber = part;
      PartDescription = description;
      Quantity = count; // validate quantity via property
      PricePerItem = price; // validate price per item via property
   } // end four-parameter Invoice constructor
```

Class `Invoice` implements interface `IPayable`. Like all classes, class `Invoice` also implicitly inherits from class `object`.

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 1 of 3.)

Invoice.cs

(2 of 3)

```

   // property that gets and sets the quantity on the invoice
   public int Quantity
   {
      get
      {
         return quantity;
      } // end get
      set
      {
         quantity = ( value < 0 ) ? 0 : value; // validate quantity
      } // end set
   } // end property Quantity

   // property that gets and sets the price per item
   public decimal PricePerItem
   {
      get
      {
         return pricePerItem;
      } // end get
```

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 2 of 3.)

Invoice.cs


```
         set
         {
            pricePerItem = ( value < 0 ) ? 0 : value; // validate price
         } // end set
      } // end property PricePerItem

      // return string representation of Invoice object
      public override string ToString()
      {
         return string.Format(
            "{0}: \n{1}: {2} ({3}) \n{4}: {5} \n{6}: {7:C}",
            "invoice", "part number", PartNumber, PartDescription,
            "quantity", Quantity, "price per item", PricePerItem );
      } // end method ToString

      // method required to carry out contract with interface IPayable
      public decimal GetPaymentAmount()
      {
         return Quantity * PricePerItem; // calculate total cost
      } // end method GetPaymentAmount
   } // end class Invoice
```

`Invoice` implements the `IPayable` interface by declaring a `GetPaymentAmount` method.

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

12.7 Case Study: Creating and Using Interfaces (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.

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

Employee.cs


```
// Fig. 12.13: Employee.cs
// Employee abstract base class.
public abstract class Employee : IPayable
{
   // read-only property that gets employee's first name
   public string FirstName { get; private set; }

   // read-only property that gets employee's last name
   public string LastName { get; private set; }

   // read-only property that gets employee's social security number
   public string SocialSecurityNumber { get; private set; }

   // three-parameter constructor
   public Employee( string first, string last, string ssn )
   {
      FirstName = first;
      LastName = last;
      SocialSecurityNumber = ssn;
   } // end three-parameter Employee constructor
```

Class `Employee` now implements interface `IPayable`.

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

Outline

Employee.cs

(2 of 2)

```
   
   // return string representation of Employee object
   public override string ToString()
   {
      return string.Format( "{0} {1}\nsocial security number: {2}",
         FirstName, LastName, SocialSecurityNumber );
   } // end method ToString

   // Note: We do not implement IPayable method GetPaymentAmount here, so
   // this class must be declared abstract to avoid a compilation error.
   public abstract decimal GetPaymentAmount();
} // end abstract class Employee
```

`Earnings` has been renamed to `GetPaymentAmount` to match the interface's requirements.

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

An implicitly implemented interface member is, by default, sealed. It must be marked virtual or abstract in the base class in order to be overridden..

- Figure 12.14 contains a modified version of class `SalariedEmployee` that extends `Employee` and implements method `GetPaymentAmount`.


```
// Fig. 12.14: SalariedEmployee.cs
// SalariedEmployee class that extends Employee.
public class SalariedEmployee : Employee
{
   private decimal weeklySalary;

   // four-parameter constructor
   public SalariedEmployee( string first, string last, string ssn,
      decimal salary ) : base( first, last, ssn )
   {
      WeeklySalary = salary; // validate salary via property
   } // end four-parameter SalariedEmployee constructor

   // property that gets and sets salaried employee's salary
   public decimal WeeklySalary
   {
      get
      {
         return weeklySalary;
      } // end get
```

Fig. 12.14 | `SalariedEmployee` class that extends `Employee`. (Part 1 of 2.)

SalariedEmployee.cs

(2 of 2)

```
      set
      {
         weeklySalary = value < 0 ? 0 : value; // validation
      } // end set
   } // end property WeeklySalary

   // calculate earnings; implement interface IPayable method
   // that was abstract in base class Employee
   public override decimal GetPaymentAmount()
   {
      return WeeklySalary;
   } // end method GetPaymentAmount

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

Method `GetPaymentAmount` replaces method `Earnings`, keeping the same functionality.

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

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

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`.

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

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

Software Engineering Observation 12.6

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.

Software Engineering Observation 12.7

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.

- `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 application.

`PayableInterfaceTest.cs`


```
// Fig. 12.15: PayableInterfaceTest.cs
// Tests interface IPayable with disparate classes.
using System;

public class PayableInterfaceTest
{
   public static void Main( string[] args )
   {
      // create four-element IPayable array
      IPayable[] payableObjects = new IPayable[ 4 ];

      // populate array with objects that implement IPayable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00M );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95M );
      payableObjects[ 2 ] = new SalariedEmployee( "John", "Smith",
         "111-11-1111", 800.00M );
      payableObjects[ 3 ] = new SalariedEmployee( "Lisa", "Barnes",
         "888-88-8888", 1200.00M );
```

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

PayableInterfaceTest.cs


```

      Console.WriteLine(
         "Invoices and Employees processed polymorphically:\n" );

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

```
Invoices and Employees processed polymorphically:

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

(continued on next page...)

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

(continued from previous page…)

```
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
```

`PayableInterfaceTest.cs`

(3 of 3)

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

Software Engineering Observation 12.8

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`.

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

- **12.7.7 Common Interfaces of the .NET Framework Class Library**

Interface	Description
`IComparable`	Objects of a class that implements the interface can be compared to one another.
`IComponent`	Implemented by any class that represents a component, including Graphical User Interface (GUI) controls.
`IDisposable`	Implemented by classes that must provide an explicit mechanism for releasing resources.
`IEnumerator`	Used for iterating through the elements of a collection (such as an array) one element at a time.

Fig. 12.16 | Common interfaces of the .NET Framework Class Library.

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

Interface features

An interface cannot contain fields or constants

An interface can contain abstract declarations of

- **Methods**
- **Properties**
- **Indexers**
- **Event declarations**

Interface methods have no access modifiers and don't use the `abstract` keyword

All interface methods must be implemented

Implemented methods must be defined as `public`

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

Interface implementation

Implemented methods may be

- ***`virtual`*** **and can be overridden in derived classes**
- **Non-** ***`virtual`*** **and cannot be overridden in derived classes**
- ***`abstract`*** **and must be overridden in derived classes**

Implemented methods must be ***`public`***

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

Interface features

An **implicitly implemented interface member** is, by default, **`sealed`**. It must be marked **`virtual`** or **`abstract`** in the base class in order to be overridden

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  public virtual void Undo()
  {
     Console.WriteLine ("TextBox.Undo");
  }
}

public class RichTextBox : TextBox
{
  public override void Undo()
  {
    Console.WriteLine ("RichTextBox.Undo");
  }
}
```

```
RichTextBox r = new RichTextBox();
r.Undo();                          // RichTextBox.Undo
((IUndoable)r).Undo();             // RichTextBox.Undo
((TextBox)r).Undo();               // RichTextBox.Undo
```

Explicit Interface Member Name Qualification

Name Hiding with Interfaces

- call a method implemented from an interface is to **cast an instance of that class to the interface type** and **then call the desired method** or **directly call the implemented method without casting the object to an interface** (*usual case, however, in the case of multiple interface inheritance quickly* ***pollute your class's public namespace*** *with members that have no meaning outside the scope of the implementing class*)
- **prevent the implemented members of interfaces from becoming public members of the** class by using a technique known as name hiding (*hide an inherited member name from any code outside the derived or implementing class*)

Explicit Interface Member Name Qualification-Example

Name Hiding with Interfaces

- an `EditBox` class needs to implement the `IDataBound` interface.
- the `EditBox` class **doesn't want to expose** the `IDataBound` methods **to the outside world** or perhaps the programmer simply doesn't want to clutter the class's namespace with a large number of methods that a typical client won't use
- To **hide an implemented interface member**, you need only remove the member's public access modifier and qualify the member name with the interface name

Note: *The **interface must be explicitly inherited** in order to hide implemented interface member*

More examples- Lab No. 8

```
using System;
public interface IDataBound
{
     void Bind();
}

public class EditBox : IDataBound
{
    // IDataBound implementation
    void IDataBound.Bind()
    {
      Console.WriteLine("Binding to data store...");
    }
}
class NameHiding2App {
  public static void Main()
  {
    Console.WriteLine();
    EditBox edit = new EditBox();
    Console.WriteLine("Calling EditBox.Bind()...");
    // ERROR: The following line won't compile because
    // the Bind method no longer exists in the
    // EditBox class's namespace.
    edit.Bind();
    Console.WriteLine();
    IDataBound bound = (IDataBound) edit;
    Console.WriteLine("Calling (IDataBound) " + "EditBox.Bind()...");
    // This is OK because the object was cast to
    // IDataBound first.
    bound.Bind();
 }
}
```

Use name hiding

InterfaceWithNameHiding.cs

Program Output

Explicit Interface Member Name Qualification

Avoiding Name Ambiguity

- C# doesn't support multiple inheritance, it does support inheritance from one class and the additional implementation of multiple interfaces.
- Problem: name collision

*Example: consider two interfaces, **ISerializable** and **IDataStore**, which support the reading and storing of data in two different formats—one as an object to disk in binary form, and the other to a database. **The problem is** that they **both contain methods named SaveData***

More examples- Lab No. 8

```
// This code wouldn't compile in future builds of C#
// The class has implemented either a serialized version
// or a database version of the SaveData method,
// but not both
using System;
interface ISerializable
  {
    void SaveData();
  }

interface IDataStore
{
    void SaveData();
}
class Test : ISerializable, IDataStore
{
    public void SaveData()
     {
       Console.WriteLine("Test.SaveData called");
     }
}
class NameCollisions1App
{
  public static void Main()
  {
    Test test = new Test();
    Console.WriteLine("Calling Test.SaveData()");
    test.SaveData();
  }
}
```

Name conflict

AvoidNameConflict1.cs

Program Output

```
// Use explicit member name qualification
using System;
interface ISerializable {
     void SaveData();
}
interface IDataStore {
     void SaveData();
}

class Test : ISerializable, IDataStore {
    void IDataStore.SaveData()      {
          Console.WriteLine("[Test.SaveData] IDataStore "
                       +            "implementation called");
    }
   void ISerializable.SaveData()      {
          Console.WriteLine("[Test.SaveData] ISerializable "
                       +            "implementation called");
    }
}
class NameCollisions3App {
    public static void Main()     {
         Test test = new Test();
         Console.WriteLine("[Main] "
             +             "Testing to see if Test implements "
             +             "ISerializable...");
         Console.WriteLine("[Main] "
             +             "ISerializable is {0}implemented",
                            test is ISerializable ? "" : "NOT ");
         ((ISerializable)test).SaveData();
         Console.WriteLine();
         Console.WriteLine("[Main] "
                +             "Testing to see if Test implements "
                +            "IDataStore...");
         Console.WriteLine("[Main] "
                +             "IDataStore is {0}implemented",
                               test is IDataStore ? "" : "NOT ");
         ((IDataStore)test).SaveData();
  }
}
```

AvoidNameConflict2.cs

Program Output

Explicit Name qualification

Inheritance with Interfaces

Problems

- **deriving from a base class** that contains a method name identical to the name of an interface method that the class needs to implement . (case A)
- a **derived class has a method with the same name** as the base class implementation of an interface method (case B)

More examples- Lab No. 8

Note: Always **cast** the object to the interface whose member you're attempting to use

```
// Deriving from a base class that contains a method name
// identical to the name of an interface method
//  that the class needs to implement
// (case A)
using System;
public class Control {
 public void Serialize()
      {
          Console.WriteLine("Control.Serialize called");
      }
 }
 public interface IDataBound
 {
     // EditBox never implements this, but it still compiles!!!
     void Serialize();
  }

public class EditBox : Control, IDataBound { }

class InterfaceInh1App
 {
   public static void Main()
     {
         EditBox edit = new EditBox();
         edit.Serialize();
     }
}
// No definition for the member in the interface's declaration
// The Control.Serialize method is being called
```

The code compiles because the C# compiler looks for an implemented Serialize method in the EditBox class and finds one. However, the compiler is incorrect in determining that this is the implemented method. The *Serialize* method found by the compiler is the Serialize method inherited from the Control class and not an actual implementation of the *IDataBound.Serialize* method

Control.Serialize method is being called

```
// (case A)
// Calling the wrong method
using System;
public class Control
 {
     public void Serialize()
      {
          Console.WriteLine("Control.Serialize called");
      }
 }
 public interface IDataBound
 {
     void Serialize();
 }
 public class EditBox : Control, IDataBound { }
 class InterfaceInh2App
 {
     public static void Main()
     {
         EditBox edit = new EditBox();
         IDataBound bound = edit as IDataBound;
     if (bound != null)
     {
         Console.WriteLine("IDataBound is supported...");
         bound.Serialize();
     }
     else
     {
         Console.WriteLine("IDataBound is NOT supported...");
     }
     }
 }
```

The following code first **checks- via the as operator-** that the interface is implemented and then attempts to call an implemented ***Serialize*** method. However, the ***EditBox*** class doesn't really implement a ***Serialize*** method as a result of the IDataBound inheritance. **The *EditBox* already had a *Serialize* method—which it inherited—from the *Control* class.** This means that the **client won't get the expected results**

The client won't get the expected results

```
// (case B)
using System;
interface ITest
 {
     void Foo();
 }
 // Base implements ITest.
 class Base : ITest {
     public void Foo()
      {
          Console.WriteLine("Base.Foo (ITest implementation)");
     }
 }
 class MyDerived : Base
 {
     public new void Foo()     {
         Console.WriteLine("MyDerived.Foo");
     }
 }
 public class InterfaceInh3App
 {
     public static void Main()  {
        Console.WriteLine("InterfaceInh3App.Main :Instantiating a MyDerived class");
        MyDerived myDerived = new MyDerived();
        Console.WriteLine();
         Console.WriteLine("InterfaceInh3App.Main :Calling MyDerived.Foo.”
          +  ”Which method will be called?");
         myDerived.Foo(); /*execute MyDerived.Foo- the new keyword make an override
                            to the inherited method */
         Console.WriteLine();
         Console.WriteLine("InterfaceInh3App.Main :Calling MyDerived.Foo:”
                           + “Casting to ITest  interface...");
         ((ITest)myDerived).Foo();//the implementation of ITest.Foo being called
     }
 }
```

Although the *myDerived* object has an inherited implementation of *ITest.Foo*, the run time will execute *MyDerived.Foo* because the *new* keyword specifies an override of the inherited method

Restrictions on Explicit Interface method implementation

You can only access methods through the `interface`

You cannot declare methods as `virtual`

(a derived class cannot access an explicit method implementation , hence cannot `override` it)

You cannot specify an access modifier(explicit method implementations have different accessibility characteristics than other methods)

You cannot access directly an explicit method implementation- only through upcasting an object to its `interface`

Restrictions on Explicit Interface method implementation

A **subclass can reimplement any interface member already implemented by a base class**.

Reimplementation hijacks a member implementation (when called through the interface) and works whether or not the member is virtual in the base class. It also works whether a member is implemented implicitly or explicitly- although it works best in the latter case, as we will demonstrate

Restrictions on Explicit Interface method implementation

In the following example, **`TextBox`** implements **`IUndoable.Undo`** explicitly, and so it cannot be marked as **`virtual`**.

In order to "**override**" it, **`RichTextBox`** must **reimplement** IUndoable's **`Undo`** method.

However, calling

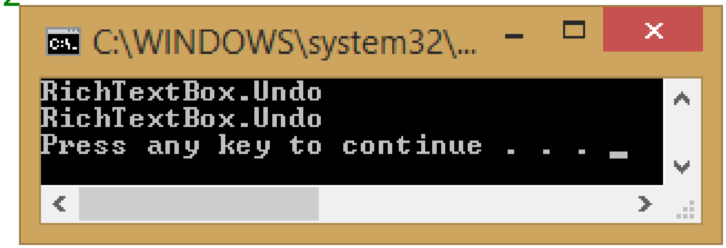
```
((TextBox)r).Undo();   // Illegal
```

Calling the reimplemented member through the interface calls the subclass's implementation.

Restrictions on Explicit Interface method implementation

An **explicitly implemented interface** member **cannot be marked virtual**, **nor can it be overridden** in the usual manner. It can, however, be **reimplemented**.

```
static void Main(string[] args)
{
     RichTextBox r = new RichTextBox();
     r.Undo();                // RichTextBox.Undo Case 1
     ((IUndoable)r).Undo(); // RichTextBox.Undo Case 2
}
public interface IUndoable { void Undo(); }
public class TextBox : IUndoable
{
     void IUndoable.Undo() => Console.WriteLine("TextBox.Undo");
}
public class RichTextBox : TextBox, IUndoable
{
    public void Undo() => Console.WriteLine("RichTextBox.Undo");
}
```



Restrictions on Explicit Interface method implementation

Assuming the same **`RichTextBox`** definition, suppose that **`TextBox`** implemented **`Undo`** *implicitly*.

Case 3 demonstrates that reimplementation **hijacking is effective** only **when a member is called through the interface** and **not through the base class**. This **is usually undesirable** as it can mean **inconsistent semantics**.

Summary:

This makes **reimplementation** most **appropriate** as a strategy for **overriding *explicitly* implemented interface members.**

Restrictions on Explicit Interface method implementation

```
C:\WINDOWS\system...
RichTextBox.Undo
RichTextBox.Undo
TextBox.Undo
Press any key to continue . . .
```

```
class Program
{
    static void Main(string[] args)
    {
        RichTextBox r = new RichTextBox();
        r.Undo(); // RichTextBox.Undo Case 1
        ((IUndoable)r).Undo(); // RichTextBox.Undo Case 2
        ((TextBox)r).Undo();    // TextBox.Undo   Case 3
    }
}
public interface IUndoable { void Undo(); }
public class TextBox : IUndoable
{
    public void Undo() { Console.WriteLine("TextBox.Undo"); }
}
public class RichTextBox : TextBox, IUndoable
{
    public new void Undo() { Console.WriteLine("RichTextBox.Undo"); }
}
```

Software Engineering Observation 12.9

Use operator overloading when it makes an application clearer than accomplishing the same operations with explicit method calls.

- C# enables you to overload most operators to make them sensitive to the context in which they are used.
- 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.

Outline

ComplexNumber.cs


```
// Fig. 12.17: ComplexNumber.cs
// Class that overloads operators for adding, subtracting
// and multiplying complex numbers.
using System;

public class ComplexNumber
{
   // read-only property that gets the real component
   public double Real { get; private set; }

   // read-only property that gets the imaginary component
   public double Imaginary { get; private set; }

   // constructor
   public ComplexNumber( double a, double b )
   {
      Real = a;
      Imaginary = b;
   } // end constructor
```

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

ComplexNumber.cs

(3 of 4)

```
    // return string representation of ComplexNumber
    public override string ToString()
    {
       return string.Format( "({0} {1} {2}i)",
          Real, ( Imaginary < 0 ? "-" : "+" ), Math.Abs( Imaginary ) );
    } // end method ToString

    // overload the addition operator
    public static ComplexNumber operator +(
       ComplexNumber x, ComplexNumber y )
    {
       return new ComplexNumber( x.Real + y.Real,
          x.Imaginary + y.Imaginary );
    } // end operator +

```

Overload the plus operator (+) to perform addition of `ComplexNumber`s

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

ComplexNumber.cs

(4 of 4)

```
   // overload the subtraction operator
   public static ComplexNumber operator -(
      ComplexNumber x, ComplexNumber y )
   {
      return new ComplexNumber( x.Real - y.Real,
         x.Imaginary - y.Imaginary );
   } // end operator -

   // overload the multiplication operator
   public static ComplexNumber operator *(
      ComplexNumber x, ComplexNumber y )
   {
      return new ComplexNumber(
         x.Real * y.Real - x.Imaginary * y.Imaginary,
         x.Real * y.Imaginary + y.Real * x.Imaginary );
   } // end operator *
} // 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`.

12.8 Operator Overloading (Cont.)

Software Engineering Observation 12.10

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 12.11

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

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


```
// Fig. 12.18: OperatorOverloading.cs
// Overloading operators for complex numbers.
using System;

public class ComplexTest
{
   public static void Main( string[] args )
   {
      // declare two variables to store complex numbers
      // to be entered by user
      ComplexNumber x, y;

      // prompt the user to enter the first complex number
      Console.Write( "Enter the real part of complex number x: " );
      double realPart = Convert.ToDouble( Console.ReadLine() );
      Console.Write(
         "Enter the imaginary part of complex number x: " );
      double imaginaryPart = Convert.ToDouble( Console.ReadLine() );
      x = new ComplexNumber( realPart, imaginaryPart );

```

Fig. 12.18 | Overloading operators for complex numbers. (Part 1 of 2.)

```
      // prompt the user to enter the second complex number
      Console.Write( "\nEnter the real part of complex number y: " );
      realPart = Convert.ToDouble( Console.ReadLine() );
      Console.Write(
         "Enter the imaginary part of complex number y: " );
      imaginaryPart = Convert.ToDouble( Console.ReadLine() );
      y = new ComplexNumber( realPart, imaginaryPart );

      // display the results of calculations with x and y
      Console.WriteLine();
      Console.WriteLine( "{0} + {1} = {2}", x, y, x + y );
      Console.WriteLine( "{0} - {1} = {2}", x, y, x - y );
      Console.WriteLine( "{0} * {1} = {2}", x, y, x * y );
   } // end method Main
} // end class ComplexTest
```

```
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)
```

`OperatorOverloading.cs`

Add, subtract and multiply `x` and `y` with the overloaded operators, then output the results.

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

12.8 Operator Overloading (Cont.)

Override the **`Equals`**() method to compare if two objects are equal

Supply **`==`** and **`!=`** operators

Override the **`GetHashCode`**() method

Overriding **`==`** by using the same algorithm as used to override **`Equals`**()

If implementing **`IComparable`**, implement Equals()

`Equals`(), **`GetHashCode`**() and the **`==`** operator never throw exceptions

12.8 Operator Overloading (Cont.)

```
public class Employee : IComparable
{
      public string name;
      public int level;
      public DateTime hiringDate;

      public Employee(string name,int
                level,DateTime hiringDate) {
                  this.name = name;
                  this.level=level;
                  this.hiringDate=hiringDate;
      }
```

12.8 Operator Overloading (Cont.)

```
public int  CompareTo(Object anObject) {// implement CompareTo() example
                          if (anObject == null) return 1;
                          if ( !(anObject is Employee) ) {
                                   throw new ArgumentException();
                          }

                          Employee anEmployee = (Employee)anObject;
                          if ( level < anEmployee.level ) return -1;
                          else {
                              if ( level == anEmployee.level ) {
                              if (hiringDate < anEmployee.hiringDate) return -1;
                                 else {
                                   if ( hiringDate == anEmployee.hiringDate)
                                                              return 0;
                                   else return 1;
                                   }
                           }
                           else return 1;
                          }
                 }
```

12.8 Operator Overloading (Cont.)

```
class Rectangle  // override Equals() example
    {
        private int x1;
        private int x2;
        private int y1;
        private int y2;

        public Rectangle(int x1, int y1, int x2, int y2)
        {
            this.x1 = x1;
            this.x2 = x2;
            this.y1 = y1;
            this.y2 = y2;
        }
        // override GetHashCode()
        public override int GetHashCode()
        {
            return x1;
        }
```

12.8 Operator Overloading (Cont.)

```
public override bool Equals(object obj)
        {
           // check for null and compare run- time types
            if (obj == null || GetType() != obj.GetType())
                                                    return false;

            Rectangle r = (Rectangle)obj;

            return (x1 == r.x1) && (y1 == r.y1) &&
                                   (x2 == r.x2) && (y2 == r.y2);

        }
```

12.8 Operator Overloading (Cont.)

```
static public bool operator ==(Rectangle r1, Rectangle r2)
        {
            // check for null parameters
            // cast to object to avoid recursive calls
            if ((object)r1 == null) return false;

            // Let Equals method handle comparison
            return r1.Equals(r2);
        }
static public bool operator !=(Rectangle r1, Rectangle r2)
        {
            // check for null parameters
            // cast to object to avoid recursive calls
            if ((object)r1 == null) return true;

            // Let Equals method handle comparison
            return !r1.Equals(r2);
        }
    }
```

User-Defined Conversion

User-defined conversions enable you to declare conversions on structures or classes so that the struct or class can be converted to other structures, classes, or basic C# types

Example:

You need to use the standard Celsius and Fahrenheit temperature scales in your application so that you can easily convert between the two

```
Fahrenheit f = 98.6F;// Implicit conversion
Celsius c = (Celsius)f;// Explicit conversion
```

User-Defined Conversion- Syntax

The syntax of the user-defined conversion **uses the *`operator`* keyword** to declare user-defined conversions:

```
public static implicit operator conv-type-out
                                   (conv-type-in operand)
public static explicit operator conv-type-out
                                   (conv-type-in operand)
```

User-Defined Conversion- rules

Any **conversion method** for a **struct** or **class—you** can define as many as you need- must be **static**.

Conversions must be defined as either **implicit** or **explicit**. The **implicit** keyword *means that the cast isn't required by the client and will* ***occur automatically***. Conversely, using the **explicit** keyword *signifies that the client must* ***explicitly cast*** *the value to the desired type*.

All conversions either **must take (as an argument) the type that the conversion is being defined** on or **must return that type.**

As with operator overloading, the `operator` keyword is used in the conversion method signature **but without any appended operator**

```
// ImplicitConversion.cs sample file
using System;
struct Celsius
 {
   public float temp;

   public Celsius(float temp)
     {
         this.temp = temp;
     }
   public static explicit operator Celsius (float temp)
     {
         Celsius c;
         c = new Celsius(temp);
         return(c);
     }
  public static implicit operator float(Celsius c)
     {
         return((((c.temp - 32) / 9) * 5));
     }
}
```

Use a constructor

```
struct Fahrenheit
 {
     public Fahrenheit(float temp)
     {
         this.temp = temp;
     }
     public static explicit operator Fahrenheit(float temp)
     {
         Fahrenheit f;
         f = new Fahrenheit(temp);
         return(f);
     }
     public static implicit operator float(Fahrenheit f)
     {
         return(((f.temp * 9) / 5) + 32));
     }
     public float temp;
 }
 class Temp1App
 {
     public static void Main()
     {
         float t;
         t=98.6F;  // implicit typecast
         Console.WriteLine("Setting {0} type to {1}", t.GetType(), t);
         Console.Write("Conversion of {0} ({1}) to Celsius = ", t.GetType(), t);
         Console.WriteLine((Celsius)t);
         Console.WriteLine();
         t=0F;
         Console.WriteLine("Setting {0} type to {1}", t.GetType(), t);
         Console.Write("Conversion of {0} ({1}) to "+"Fahrenheit =", t.GetType(), t);
         Console.WriteLine((Fahrenheit)t);
         Console.ReadLine();
     }
 }
```

Use implicit conversion

User-Defined Conversion- problems and solutions

You can pass only values of type float to these conversion methods

```
// ERROR: This code will not compile in
// Temp1App because
// there is no explicit conversion method from Celsius
// to Fahrenheit defined.
    Celsius c = new Celsius(55);
Console.WriteLine((Fahrenheit)c);
```

```
using System;
class Temperature
 {
     protected float temp;

     public Temperature(float Temp)
     {
         this.temp = Temp;
     }

     public float Temp
     {
         get
         {
            return this.temp;
         }
     }
 }
 class Celsius : Temperature
 {
     public Celsius(float Temp): base(Temp) {}

     public static implicit operator Celsius(float Temp)
     {
         return new Celsius(Temp);
     }
     public static  explicit  operator Celsius(Fahrenheit F)
     {
         return new Celsius(F.temp);
     }
     public static implicit operator float(Celsius C)     {
         return(((C.temp - 32) / 9) * 5));
     }
 } x
```

Changed the *`Celsius`* and *`Fahrenheit`* types from *`struct`* to *`class`*. A practical reason for making this change is to share the *`temp`* member variable by having the *`Celsius`* and *`Fahrenheit`* classes derive from the same *`Temperature`* base class

Outline

```
class Fahrenheit : Temperature
 {
     public Fahrenheit(float Temp): base(Temp) {}
     public static implicit operator Fahrenheit(float Temp)
     {
         return new Fahrenheit(Temp);
     }
     public static explicit operator Fahrenheit(Celsius C)
     {
         return new Fahrenheit(C.temp);
     }
     public static implicit operator float(Fahrenheit F)
     {          return((((F.temp * 9) / 5) + 32));
     }
 }
 class Temp2App
 {
     public static void DisplayTemp(Celsius Temp)
     {
         Console.Write("Conversion of {0} {1} to Fahrenheit =",
                                          Temp.ToString(), Temp.Temp);
         Console.WriteLine((Fahrenheit)Temp);
     }
     public static void DisplayTemp(Fahrenheit Temp)
     {
         Console.Write("Conversion of {0} {1} to Celsius =",
                                          Temp.ToString(), Temp.Temp);
         Console.WriteLine((Celsius)Temp);
     }
     public static void Main()
     {
         Fahrenheit f = new Fahrenheit(98.6F);
         DisplayTemp(f);
         Celsius c = new Celsius(0F);
         DisplayTemp(c);
         Console.ReadLine();
     }
 }
```

Structs

A **structure can have its own fields, methods, and constructors** just like a class, but not like an enumeration

In C#, the **primitive numeric types** *int, long, and float are aliases for* the structures *System.Int32, System.Int64, and System.Single, respectively*. ***These structures* have fields and methods**, and you can actually call methods on variables and literals of these types.

For example, all of these structures provide a *ToString method that* can convert a numeric value to its string representation. The **following statements are all legal statements** in C#:

```
int i = 99;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
float f = 98.765F;
    string s = "42";
    int i = int.Parse(s);
// exactly the same as Int32.Parse
```

Structs

To declare your own structure value type, you use the *struct keyword followed by the name* of the type, followed by the body of the structure between opening and closing braces.

For example, here is a structure named *Time that contains three public int fields named hours, minutes, and seconds.*

As with classes, making the fields of a structure *public is not advisable in most cases; there* is no way to ensure that *public fields contain valid values.*
For example, anyone could set the value of *minutes or seconds to a value greater than 60.*

A better idea is to make the fields private and provide your structure with constructors and methods to initialize and manipulate these fields

Structs

```
struct Time
{
   private int hours, minutes, seconds;
   public Time(int hh, int mm, int ss)
   {
     hours   = hh % 24;
     minutes = mm % 60;
     seconds = ss % 60;
   }
   public int Hours()
   {
     return hours;
   }
   ...
}
```

Structs

A structure can have its own fields, methods, and constructors just like a class, but not like an enumeration

Structs vs. Classes

Similar to classes, but there are important differences that you should be aware of.

First of all, classes are reference types and structs are **value types**. By using **structs**, you can create **objects that behave like the built-in types** and enjoy their benefits as well

Heap or Stack

When you call the **new** operator on a **class, it will be allocated on the heap**. However, when you instantiate a **struct**, it **gets created on the stack**. This **will yield performance gains**. Also, you will not be dealing with references to an instance of a **struct** as you would with classes. You will be working directly with the **struct** instance. Because of this, when passing a **struct** to a method, it's **passed by value instead of as a reference**.

Structs

```
using System;
class TheClass
{
    public int x;
}
struct TheStruct
{
    public int x;
}
class TestClass
{
    public static void structtaker(TheStruct s)
    {
        s.x = 5;
    }
    public static void classtaker(TheClass c)
    {
        c.x = 5;
    }
```

Structs

An important design goal is **to keep `structs` immutable**.

As demonstrated on the previous slide `structs` are **copied by value**, not **by reference**. If you **make a change** to a `struct`, **you might actually be modifying a copy**. Therefore it is easy to accidentally treat a `struct` as being copied by reference.

Additionally, an **immutable `struct`** is **inherently thread-safe**.

Examples for immutable `structs`- `int`, `double`, `bool`.

Structs

```
// continues class TheClassTest
public static void Main()
{
    TheStruct a = new TheStruct();
    TheClass  b = new TheClass();
    a.x = 1;
    b.x = 1;
    structtaker(a);
    classtaker(b);
    Console.WriteLine("a.x = {0}", a.x);
    Console.WriteLine("b.x = {0}", b.x);
}
}
```

Output

a.x = 1

b.x = 5

Structs

Constructors and Inheritance

Struct can **declare only general purpose constructors**. It is an error to declare a default constructor for a ***struct***.

Struct members cannot have initializers. A **default constructor is always provided** to initialize the ***struct*** members to their default values.

When you create a ***struct*** object using the ***new*** operator, it gets created and the appropriate constructor is called.

Structs

```
struct Time
{
        public Time() { ... } // compile-time error
        ...
}
struct Time
{
        private int hours, minutes, seconds;
        public Time(int hh, int mm)
        {
                hours = hh;
                minutes = mm;
        } // compile-time error: seconds not initialized
...
}
```

Structs

```
struct Time
{
    ...
    private int hours = 0; // compile-time error
    private int minutes;
    private int seconds;
}
```

Structs

Unlike classes, **`structs`** can be instantiated without using the **`new`** operator. If you do not use **`new`**, the fields will remain unassigned and the **object cannot be used until all the fields are initialized**.

There is no inheritance for ***`structs`*** as there is for classes. A ***`struct`*** **cannot inherit** from another ***`struct`*** or class, and it **cannot be the *`base`*** of a class. ***`Structs`***, however, **inherit from the base class *`object`***. A ***`struct`*** can implement ***`interfaces`***, and it does that exactly as classes do.

Structs

```
interface IImage
{
    void Paint();
}

struct Picture : IImage
{
  private int x, y, z;
  // other struct members
    public void Paint()
    {
          // painting code goes here
    }
}
```

Structs

```
struct MyStruct
{
    public MyStruct ( int size )
    {
        this.Size = size; // <-- Compile-Time Error!
    }

    public int Size{get; set;}
}
```

You need to **call the default constructor** for this to work

```
public MyStruct(int size) : this()
{
    Size = size;
}
```

```
public int Size {get; private set;}
```

Structs

```
struct Mutable {
    private int x;
    public int Mutate() {
        this.x = this.x + 1;
        return this.x;
    }
}
class Test {
    public readonly Mutable m = new Mutable();
    static void Main(string[] args) {
        Test t = new Test();
        System.Console.WriteLine(t.m.Mutate());
        System.Console.WriteLine(t.m.Mutate());
        System.Console.WriteLine(t.m.Mutate());
    }
}
```

1) Print 1, 2, 3 — because m is readonly, but the "readonly" only applies to m, not to its contents.
2) Print 0, 0, 0 — because m is readonly, x cannot be changed. It always has its default value of zero.
3) Throw an exception at runtime, when the attempt is made to mutate the contents of a readonly field.
4) Do something else

Structs

In fact, this prints `1, 1, 1`.

Explanation.

Note 1: The `struct Mutable` is a value type and accessing a value type gives you a COPY of the value. When you say `t.m`, you get a copy of whatever is presently stored in `m`. `m` is immutable, but the copy is not immutable. The copy is then mutated, and the value of `x` in the copy is returned. But `m` remains untouched.

When resolving "`E.I`" where `E` is an object and `I` is a field the following rule applies. If the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field `I` in the object referenced by `E`.

The important word here is that the result is the value of the field, not the variable associated with the field. `Readonly` fields are not variables outside of the constructor. (The initializer here is considered to be inside the constructor)

Structs

Note 2: What about that second dot, as in "`.Mutate()`"? We have to explain how to invoke `E.M()`.

If `E` is not classified as a variable, then a temporary local variable of `E`'s type is created and the value of `E` is assigned to that variable. `E` is then reclassified as a reference to that temporary local variable. The temporary variable is accessible as `this` within `M`, but not in any other way. Thus, only when `E` is a `true` variable is it possible for the caller to observe the changes that `M` makes to `this`.

Conclusion:

This is yet another reason why mutable value types are evil. Try to always make value types immutable.

Question: What would be the output of the program, if `Mutable` were `class` instead of a `struct`?

The Nullable type and the coalesting operator

Nullable types allow you to create a value type variable that can be marked as valid or invalid, effectively letting you set a value type variable to “**`null`**.”

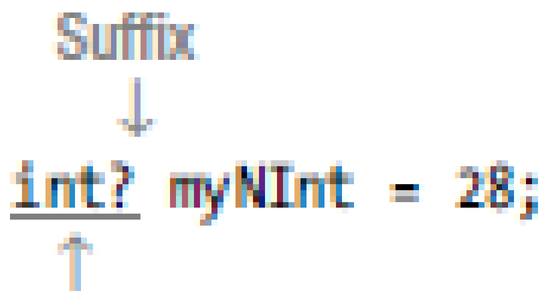
A nullable type is always based on another type, called the *underlying type*, that has already been declared. You can create a nullable type from any value type, including the predefined, simple types.

- You **cannot** create a nullable type **from a reference type or another nullable type.**
- You do not explicitly declare a nullable type in your code. Instead, you **declare** a *variable of a nullable type*. The **compiler implicitly creates** the **`nullable`** type for you

The Nullable type and the coalesting operator

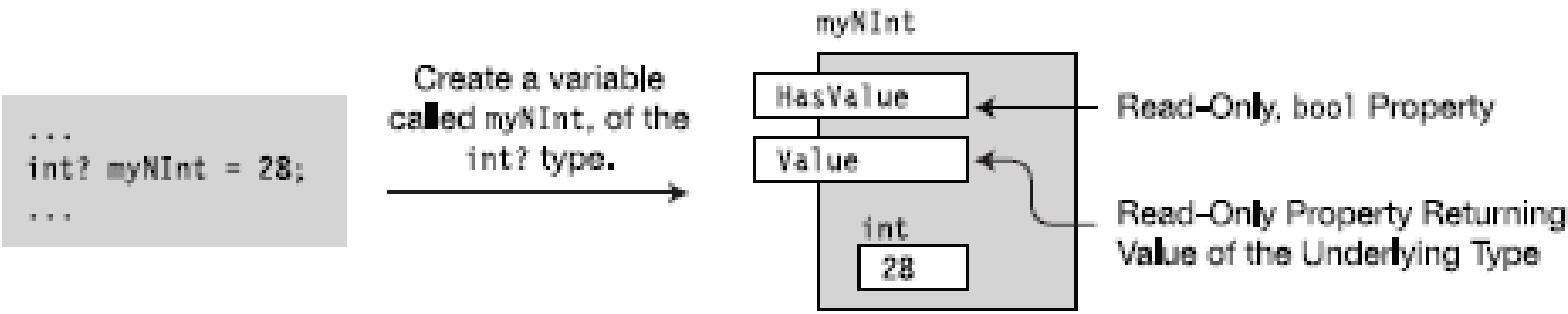
To create a variable of a nullable type, simply add a question mark to the end of the name of the underlying type, in the variable declaration.

For example, the following code declares a variable of the nullable int type. **Notice** that the **suffix is attached to the *type* name- not the variable name**



The Nullable type and the coalesting operator

The compiler **takes care of both** producing **the nullable type** and **the variable of that type**



The Nullable type and the coalesting operator

Using a nullable type is almost the same as using a variable of any other type. Reading a variable of a nullable type returns its value. You must, however, make sure that the variable is not null. **Attempting to read the value of a null variable produces an exception**.

```
int? myInt1 = 15;
        Compare to null.

if ( myInt1 != null )
   Console.WriteLine("{0}", myInt1);

                   Use variable name.
```

The Nullable type and the coalesting operator

The Null Coalescing Operator

The standard **arithmetic and comparison** operators also handle **nullable** types. There is also a special operator called the **null coalescing operator**, which **returns** a **non-null value** to an expression, in case a nullable type variable is `null`.

The null coalescing operator consists of **two contiguous question marks** and has **two operands**.

- The **first operand** is a variable of a **nullable type**.
- The **second** is a **non-nullable value** of the underlying type.
- If, at run time, the first operand (the nullable operand) evaluates to `null`, the **nonnullable operand is returned** as the result of the expression.

The Nullable type and the coalesting operator

Null coalescing operator

```
int? myI4 = null;
Console.WriteLine("myI4: {0}", myI4 ?? -1);

myI4 = 10;
Console.WriteLine("myI4: {0}", myI4 ?? -1);
```

This code produces the following output:

```
myI4: -1
myI4: 10
```

The Nullable type and the coalesting operator

You can also create `nullable` variables of user-defined value type as `struct`. The main issue is **access to the members of the encapsulated underlying type**. A `nullable` type doesn't directly expose any of the members of the underlying type.

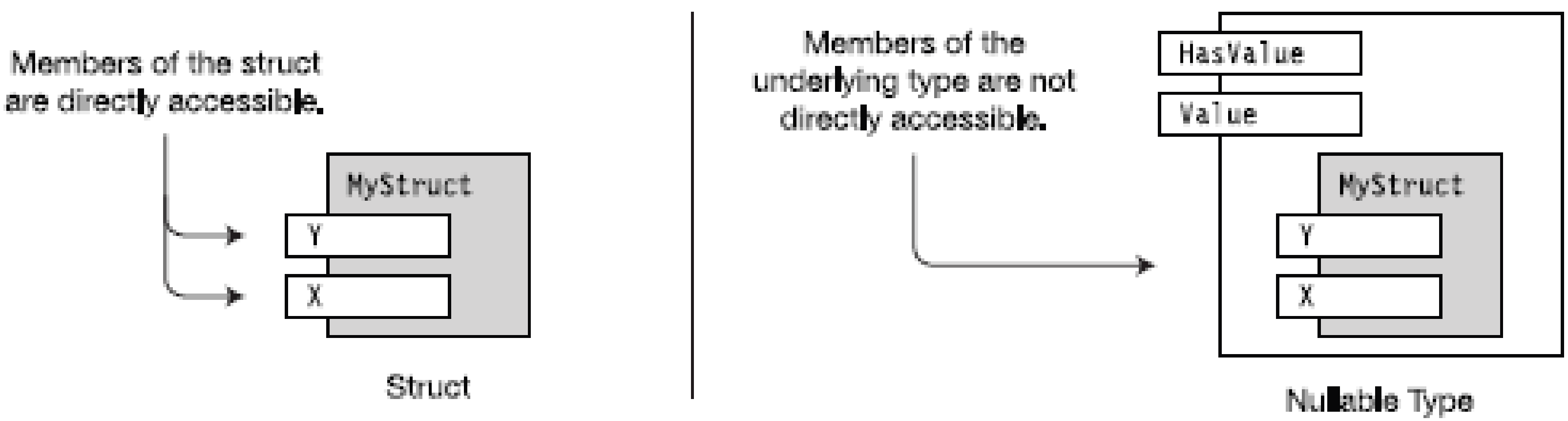
Since the fields of the `struct` are `public`, they can easily be accessed in any instance of the `struct`, as shown on the left of the figure.

- The `nullable` version of the `struct`, however, **exposes the underlying type only through** the `Value` property and **doesn't *directly* expose any of its members**.
- Although the members are `public` to the `struct`, they are not `public` to the `nullable` type

The Nullable type and the coalesting operator

```
struct MyStruct                                    // Declare a struct.
{
   public int X;                                   // Field
   public int Y;                                   // Field
   public MyStruct(int xVal, int yVal)             // Constructor
   { X = xVal;  Y = yVal; }
}

class Program {
   static void Main()
   {
      MyStruct? mSNull = new MyStruct(5, 10);
      ...
```



The Nullable type and the coalesting operator

```
MyStruct  mSStruct = new MyStruct(6, 11);      // Variable of struct
MyStruct? mSNull   = new MyStruct(5, 10);      // Variable of nullable type
                                        Struct access

Console.WriteLine("mSStruct.X: {0}", mSStruct.X);
Console.WriteLine("mSStruct.Y: {0}", mSStruct.Y);

Console.WriteLine("mSNull.X: {0}",   mSNull.Value.X);
Console.WriteLine("mSNull.Y: {0}",   mSNull.Value.Y);

                                       Nullable type access
```

The Nullable type and the coalesting operator

`Nullable<T>`

Nullable types are implemented by using a .NET type called **`System.Nullable<T>`**, which uses the C# generics feature. The **question mark syntax** of C# nullable types is **just shortcut syntax** for creating a variable of type **`Nullable<T>`**, where **`T`** is the underlying **value** type.

`Nullable<T>` takes the underlying type, embeds it in a structure, and provides the structure with the properties, methods, and constructors of the **`nullable`** type.

The Nullable type and the coalesting operator

`Nullable<T>`

```
Nullable<MyStruct> mSNull = new Nullable<MyStruct>();
```

and

```
MyStruct? mSNull = new MyStruct();
```

are equivalent semantically