OOP: Polymorphism and Interfaces

Chapter 12 of Visual C# How to Program, 6/e

OBJECTIVES

In this chapter you'll:

- Understand how polymorphism enables you to "program in the general" and make systems extensible.
- Use overridden methods to effect polymorphism.
- Create abstract classes and methods.
- Determine an object's type at execution time with operator `is`, then use downcasting to perform type-specific processing.
- Create `sealed` methods and classes.
- Declare and implement interfaces.
- Be introduced to interfaces `IComparable`, `IComponent`, `IDisposable` and `IEnumerator` of the .NET Framework Class Library.

12.1 Introduction

- **Polymorphism** enables you to write apps that process objects that share the same base class in a class hierarchy as if they were all objects of the base class.
- Polymorphism promotes 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 `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`.
- The polymorphism occurs when an app invokes a method through a base-class variable.

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 app 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.

Software Engineering Observation 12.1

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the polymorphic system logic. 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.

Software Engineering Observation 12.2

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 app needs to perform a derived-class-specific operation on a derived-class object referenced by a base-class variable, the app must first cast the base-class reference to a derived-class reference through a technique known as downcasting. This enables the app to invoke derived-class methods that are not in the base class.
- Fig. 12.1 demonstrates three ways to use base-class and derived-class variables.

```
1  // Fig. 12.1: PolymorphismTest.cs
2  // Assigning base-class and derived-class references to base-class and
3  // derived-class variables.
4  using System;
5
6  class PolymorphismTest
7  {
8     static void Main()
9     {
10       // assign base-class reference to base-class variable
11       var commissionEmployee = new CommissionEmployee(
12          "Sue", "Jones", "222-22-2222", 10000.00M, .06M);
13
14       // assign derived-class reference to derived-class variable
15       var basePlusCommissionEmployee = new BasePlusCommissionEmployee(
16          "Bob", "Lewis", "333-33-3333", 5000.00M, .04M, 300.00M);
17
```

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 1 of 5.)

```
18          // invoke ToString and Earnings on base-class object
19          // using base-class variable
20          Console.WriteLine(
21             "Call CommissionEmployee's ToString and Earnings methods " +
22             "with base-class reference to base class object\n");
23          Console.WriteLine(commissionEmployee.ToString());
24          Console.WriteLine($"earnings: {commissionEmployee.Earnings()}\n");
25
26          // invoke ToString and Earnings on derived-class object
27          // using derived-class variable
28          Console.WriteLine("Call BasePlusCommissionEmployee's ToString and" +
29             " Earnings methods with derived class reference to" +
30             " derived-class object\n");
31          Console.WriteLine(basePlusCommissionEmployee.ToString());
32          Console.WriteLine(
33             $"earnings: {basePlusCommissionEmployee.Earnings()}\n");
```

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 2 of 5.)

```
34
35          // invoke ToString and Earnings on derived-class object
36          // using base-class variable
37          CommissionEmployee commissionEmployee2 = basePlusCommissionEmployee;
38          Console.WriteLine(
39             "Call BasePlusCommissionEmployee's ToString and Earnings " +
40             "methods with base class reference to derived-class object");
41          Console.WriteLine(commissionEmployee2.ToString());
42          Console.WriteLine(
43             $"earnings: {basePlusCommissionEmployee.Earnings()}\n");
44       }
45    }
```

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 3 of 5.)

```
Call CommissionEmployee's ToString and Earnings methods 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 BasePlusCommissionEmployee's ToString and Earnings methods 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
```

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 4 of 5.)

```
Call BasePlusCommissionEmployee's ToString and Earnings methods 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
```

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 5 of 5.)

12.3 Demonstrating Polymorphic Behavior (Cont.)

- When the compiler encounters a `virtual` method call made through a variable, the compiler checks the variable's class type to determines if the method can be called.
- At execution time, the type of the object to which the variable refers determines the actual method to use.

Software Engineering Observation 12.3

A base-class variable that contains a reference to a derived-class object and is used to call a `virtual` *method actually calls the overriding derived-class version of the method.*

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 (non-`abstract`) 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; }`
- An `abstract` property omits implementations for the `get` accessor and/or the `set` accessor.
- 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` or `virtual`

Software Engineering Observation 12.4

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. The instance variables, concrete methods and concrete properties of an abstract class are subject to the normal rules of inheritance.

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.

12.5 Case Study: Payroll System Using Polymorphism

- 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 "time-and-a-half" 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. The company wants to implement an app that performs its payroll calculations polymorphically.

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 app.

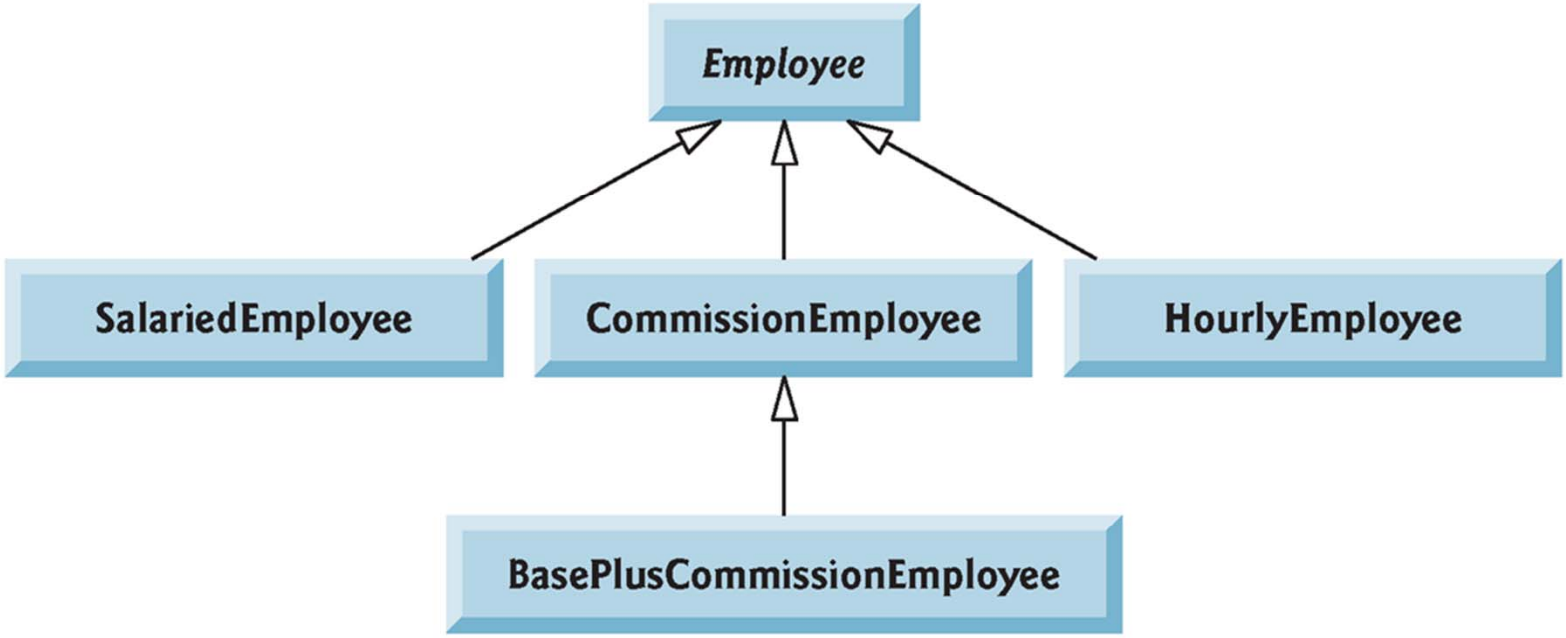

Fig. 12.2 | `Employee` hierarchy UML class diagram.

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 app iterates through the array and calls method `Earnings` for each `Employee` object. These method calls are processed 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.
- The `Employee` class's declaration is shown in Fig. 12.4.

	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	baseSalary + (commissionRate * grossSales)	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.

```
1  // Fig. 12.4: Employee.cs
2  // Employee abstract base class.
3  public abstract class Employee
4  {
5     public string FirstName { get; }
6     public string LastName { get; }
7     public string SocialSecurityNumber { get; }
8
9     // three-parameter constructor
10    public Employee(string firstName, string lastName,
11       string socialSecurityNumber)
12    {
13       FirstName = firstName;
14       LastName = lastName;
15       SocialSecurityNumber = socialSecurityNumber;
16    }
17
18    // return string representation of Employee object, using properties
19    public override string ToString() => $"{FirstName} {LastName}\n" +
20       $"social security number: {SocialSecurityNumber}";
21
22    // abstract method overridden by derived classes
23    public abstract decimal Earnings(); // no implementation here
24 }
```

Fig. 12.4 | Employee abstract base class.

12.5.2 Creating Concrete Derived Class SalariedEmployee

- The `SalariedEmployee` class's declaration is shown in Fig. 12.5.

```
1  // Fig. 12.5: SalariedEmployee.cs
2  // SalariedEmployee class that extends Employee.
3  using System;
4
5  public class SalariedEmployee : Employee
6  {
7     private decimal weeklySalary;
8
9     // four-parameter constructor
10    public SalariedEmployee(string firstName, string lastName,
11       string socialSecurityNumber, decimal weeklySalary)
12       : base(firstName, lastName, socialSecurityNumber)
13    {
14       WeeklySalary = weeklySalary; // validate salary
15    }
16
```

Fig. 12.5 | SalariedEmployee class that extends Employee. (Part 1 of 3.)

```
17      // property that gets and sets salaried employee's salary
18      public decimal WeeklySalary
19      {
20         get
21         {
22            return weeklySalary;
23         }
24         set
25         {
26            if (value < 0) // validation
27            {
28               throw new ArgumentOutOfRangeException(nameof(value),
29                  value, $"{nameof(WeeklySalary)} must be >= 0");
30            }
31
32            weeklySalary = value;
33         }
34      }
```

Fig. 12.5 | `SalariedEmployee` class that extends `Employee`. (Part 2 of 3.)

```
35
36      // calculate earnings; override abstract method Earnings in Employee
37      public override decimal Earnings() => WeeklySalary;
38
39      // return string representation of SalariedEmployee object
40      public override string ToString() =>
41         $"salaried employee: {base.ToString()}\n" +
42         $"weekly salary: {WeeklySalary:C}";
43   }
```

Fig. 12.5 | `SalariedEmployee` class that extends `Employee`. (Part 3 of 3.)

12.5.3 Creating Concrete Derived Class HourlyEmployee

- The `HourlyEmployee` class's declaration is shown in Fig. 12.6.

```
1  // Fig. 12.6: HourlyEmployee.cs
2  // HourlyEmployee class that extends Employee.
3  using System;
4
5  public class HourlyEmployee : Employee
6  {
7     private decimal wage; // wage per hour
8     private decimal hours; // hours worked for the week
9
10    // five-parameter constructor
11    public HourlyEmployee(string firstName, string lastName,
12       string socialSecurityNumber, decimal hourlyWage,
13       decimal hoursWorked)
14       : base(firstName, lastName, socialSecurityNumber)
15    {
16       Wage = hourlyWage; // validate hourly wage
17       Hours = hoursWorked; // validate hours worked
18    }
19
```

Fig. 12.6 | HourlyEmployee class that extends Employee. (Part 1 of 4.)

```
20      // property that gets and sets hourly employee's wage
21      public decimal Wage
22      {
23         get
24         {
25            return wage;
26         }
27         set
28         {
29            if (value < 0) // validation
30            {
31               throw new ArgumentOutOfRangeException(nameof(value),
32                  value, $"{nameof(Wage)} must be >= 0");
33            }
34
35            wage = value;
36         }
37      }
38
```

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 2 of 4.)

```
39      // property that gets and sets hourly employee's hours
40      public decimal Hours
41      {
42         get
43         {
44            return hours;
45         }
46         set
47         {
48            if (value < 0 || value > 168) // validation
49            {
50               throw new ArgumentOutOfRangeException(nameof(value),
51                  value, $"{nameof(Hours)} must be >= 0 and <= 168");
52            }
53
54            hours = value;
55         }
56      }
57
```

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 3 of 4.)

```
58      // calculate earnings; override Employee's abstract method Earnings
59      public override decimal Earnings()
60      {
61         if (Hours <= 40) // no overtime
62         {
63            return Wage * Hours;
64         }
65         else
66         {
67            return (40 * Wage) + ((Hours - 40) * Wage * 1.5M);
68         }
69      }
70
71      // return string representation of HourlyEmployee object
72      public override string ToString() =>
73         $"hourly employee: {base.ToString()}\n" +
74         $"hourly wage: {Wage:C}\nhours worked: {Hours:F2}";
75   }
```

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 4 of 4.)

12.5.4 Creating Concrete Derived Class CommissionEmployee

- The `CommissionEmployee` class's declaration is shown in Fig. 12.7.

```
1  // Fig. 12.7: CommissionEmployee.cs
2  // CommissionEmployee class that extends Employee.
3  using System;
4
5  public class CommissionEmployee : Employee
6  {
7     private decimal grossSales; // gross weekly sales
8     private decimal commissionRate; // commission percentage
9
10    // five-parameter constructor
11    public CommissionEmployee(string firstName, string lastName,
12       string socialSecurityNumber, decimal grossSales,
13       decimal commissionRate)
14       : base(firstName, lastName, socialSecurityNumber)
15    {
16       GrossSales = grossSales; // validates gross sales
17       CommissionRate = commissionRate; // validates commission rate
18    }
19
```

Fig. 12.7 | CommissionEmployee class that extends Employee. (Part 1 of 4.)

```
20      // property that gets and sets commission employee's gross sales
21      public decimal GrossSales
22      {
23         get
24         {
25            return grossSales;
26         }
27         set
28         {
29            if (value < 0) // validation
30            {
31               throw new ArgumentOutOfRangeException(nameof(value),
32                  value, $"{nameof(GrossSales)} must be >= 0");
33            }
34
35            grossSales = value;
36         }
37      }
38
```

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 2 of 4.)

```
39      // property that gets and sets commission employee's commission rate
40      public decimal CommissionRate
41      {
42         get
43         {
44            return commissionRate;
45         }
46         set
47         {
48            if (value <= 0 || value >= 1) // validation
49            {
50               throw new ArgumentOutOfRangeException(nameof(value),
51                  value, $"{nameof(CommissionRate)} must be > 0 and < 1");
52            }
53
54            commissionRate = value;
55         }
56      }
57
```

Fig. 12.7 | CommissionEmployee class that extends Employee. (Part 3 of 4.)

```
58      // calculate earnings; override abstract method Earnings in Employee
59      public override decimal Earnings() => CommissionRate * GrossSales;
60
61      // return string representation of CommissionEmployee object
62      public override string ToString() =>
63         $"commission employee: {base.ToString()}\n" +
64         $"gross sales: {GrossSales:C}\n" +
65         $"commission rate: {CommissionRate:F2}";
66   }
```

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 4 of 4.)

12.5.5 Creating Indirect Concrete Derived Class BasePlusCommissionEmployee

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

```
1  // Fig. 12.8: BasePlusCommissionEmployee.cs
2  // BasePlusCommissionEmployee class that extends CommissionEmployee.
3  using System;
4
5  public class BasePlusCommissionEmployee : CommissionEmployee
6  {
7     private decimal baseSalary; // base salary per week
8
9     // six-parameter constructor
10    public BasePlusCommissionEmployee(string firstName, string lastName,
11       string socialSecurityNumber, decimal grossSales,
12       decimal commissionRate, decimal baseSalary)
13       : base(firstName, lastName, socialSecurityNumber,
14           grossSales, commissionRate)
15    {
16       BaseSalary = baseSalary; // validates base salary
17    }
18
```

Fig. 12.8 | BasePlusCommissionEmployee class that extends CommissionEmployee. (Part 1 of 3.)

```
19      // property that gets and sets
20      // BasePlusCommissionEmployee's base salary
21      public decimal BaseSalary
22      {
23         get
24         {
25            return baseSalary;
26         }
27         set
28         {
29            if (value < 0) // validation
30            {
31               throw new ArgumentOutOfRangeException(nameof(value),
32                  value, $"{nameof(BaseSalary)} must be >= 0");
33            }
34
35            baseSalary = value;
36         }
37      }
38
```

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 2 of 3.)

```
39      // calculate earnings
40      public override decimal Earnings() => BaseSalary + base.Earnings();
41
42      // return string representation of BasePlusCommissionEmployee
43      public override string ToString() =>
44         $"base-salaried {base.ToString()}\nbase salary: {BaseSalary:C}";
45   }
```

Fig. 12.8 | BasePlusCommissionEmployee class that extends CommissionEmployee. (Part 3 of 3.)

12.5.6 Polymorphic Processing, Operator is and Downcasting

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

```
1  // Fig. 12.9: PayrollSystemTest.cs
2  // Employee hierarchy test app.
3  using System;
4  using System.Collections.Generic;
5
6  class PayrollSystemTest
7  {
8     static void Main()
9     {
10       // create derived-class objects
11       var salariedEmployee = new SalariedEmployee("John", "Smith",
12          "111-11-1111", 800.00M);
13       var hourlyEmployee = new HourlyEmployee("Karen", "Price",
14          "222-22-2222", 16.75M, 40.0M);
15       var commissionEmployee = new CommissionEmployee("Sue", "Jones",
16          "333-33-3333", 10000.00M, .06M);
17       var basePlusCommissionEmployee =
18          new BasePlusCommissionEmployee("Bob", "Lewis",
19          "444-44-4444", 5000.00M, .04M, 300.00M);
```

Fig. 12.9 | Employee hierarchy test app. (Part 1 of 7.)

```
20
21          Console.WriteLine("Employees processed individually:\n");
22
23          Console.WriteLine($"{salariedEmployee}\nearned: " +
24              $"{salariedEmployee.Earnings():C}\n");
25          Console.WriteLine(
26             $"{hourlyEmployee}\nearned: {hourlyEmployee.Earnings():C}\n");
27          Console.WriteLine($"{commissionEmployee}\nearned: " +
28              $"{commissionEmployee.Earnings():C}\n");
29          Console.WriteLine($"{basePlusCommissionEmployee}\nearned: " +
30              $"{basePlusCommissionEmployee.Earnings():C}\n");
31
32          // create List<Employee> and initialize with employee objects
33          var employees = new List<Employee>() {salariedEmployee,
34             hourlyEmployee, commissionEmployee, basePlusCommissionEmployee};
35
```

Fig. 12.9 | Employee hierarchy test app. (Part 2 of 7.)

```
36          Console.WriteLine("Employees processed polymorphically:\n");
37
38          // generically process each element in employees
39          foreach (var currentEmployee in employees)
40          {
41             Console.WriteLine(currentEmployee); // invokes ToString
42
43             // determine whether element is a BasePlusCommissionEmployee
44             if (currentEmployee is BasePlusCommissionEmployee)
45             {
46                // downcast Employee reference to
47                // BasePlusCommissionEmployee reference
48                var employee = (BasePlusCommissionEmployee) currentEmployee;
49
50                employee.BaseSalary *= 1.10M;
51                Console.WriteLine("new base salary with 10% increase is: " +
52                   $"{employee.BaseSalary:C}");
53             }
54
55             Console.WriteLine($"earned: {currentEmployee.Earnings():C}\n");
56          }
```

Fig. 12.9 | Employee hierarchy test app. (Part 3 of 7.)

```
57
58          // get type name of each object in employees
59          for (int j = 0; j < employees.Count; j++)
60          {
61             Console.WriteLine(
62                $"Employee {j} is a {employees[j].GetType()}");
63          }
64       }
65    }
```

Fig. 12.9 | Employee hierarchy test app. (Part 4 of 7.)

```
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

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
```

Fig. 12.9 | `Employee` hierarchy test app. (Part 5 of 7.)

```
Employees processed polymorphically:

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

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
```

Fig. 12.9 | `Employee` hierarchy test app. (Part 6 of 7.)

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

Fig. 12.9 | `Employee` hierarchy test app. (Part 7 of 7.)

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 12.5

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's 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.*

12.5.6 Polymorphic Processing, Operator `is` and Downcasting (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.

12.5.7 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 (and non-`virtual` methods) are resolved at compile time—this is known as static binding.

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

Common Programming Error 12.4

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

12.7 Case Study: Creating and Using Interfaces

- Interfaces define and standardize the ways in which people and systems can interact with one another.
- A C# interface describes a set of methods that can be called on an object—to tell it, for example, to perform some task or return some piece of information.
- An interface declaration begins with the keyword `interface` and can contain only `abstract` methods, `abstract` properties, `abstract` indexers and `abstract` events
- All interface members are implicitly declared `public` and `abstract`.
- An interface can extend one or more other interfaces to create a more elaborate interface that other classes can implement.

Common Programming Error 12.5

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

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 12.6

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

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

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

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

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

12.7.1 Developing an IPayable Hierarchy

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

Good Programming Practice 12.1

By convention, the name of an interface begins with `I` *(e.g.,* `IPayable`*). This helps distinguish interfaces from classes, improving code readability.*

Good Programming Practice 12.2

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

12.7.1 Developing an IPayable Hierarchy (Cont.)

UML Diagram Containing an Interface

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

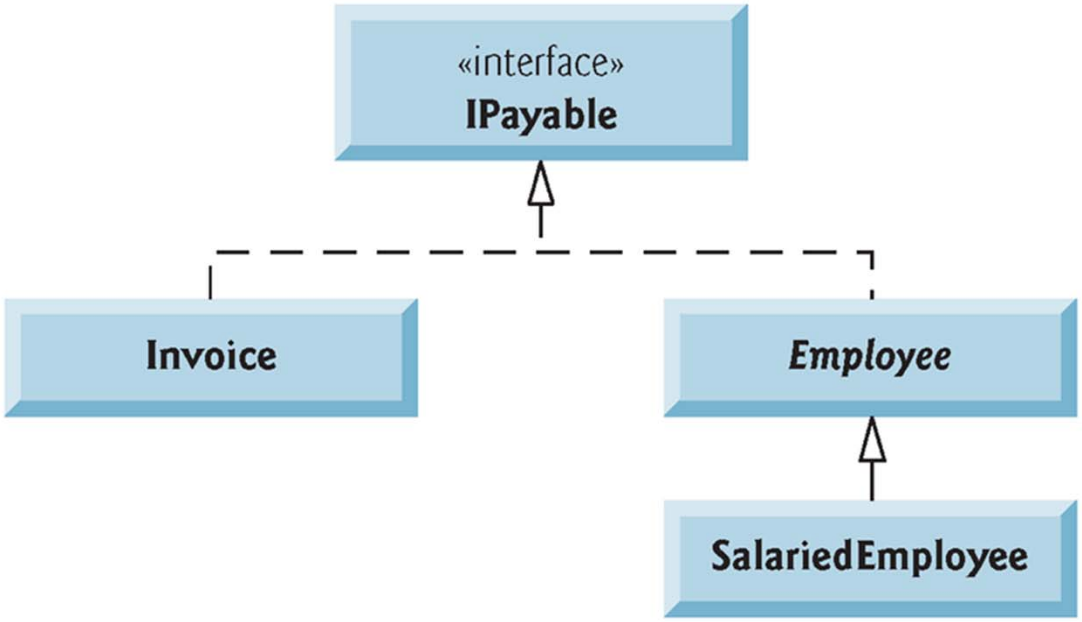

Fig. 12.10 | `IPayable` interface and class hierarchy UML class diagram.

12.7.2 Declaring Interface IPayable

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

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

Fig. 12.11 | `IPayable` interface declaration.

12.7.3 Creating Class Invoice

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

```
1  // Fig. 12.12: Invoice.cs
2  // Invoice class implements IPayable.
3  using System;
4
5  public class Invoice : IPayable
6  {
7     public string PartNumber { get; }
8     public string PartDescription { get; }
9     private int quantity;
10    private decimal pricePerItem;
11
12    // four-parameter constructor
13    public Invoice(string partNumber, string partDescription, int quantity,
14       decimal pricePerItem)
15    {
16       PartNumber = partNumber;
17       PartDescription = partDescription;
18       Quantity = quantity; // validate quantity
19       PricePerItem = pricePerItem; // validate price per item
20    }
21
```

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

```
22      // property that gets and sets the quantity on the invoice
23      public int Quantity
24      {
25         get
26         {
27            return quantity;
28         }
29         set
30         {
31            if (value < 0) // validation
32            {
33               throw new ArgumentOutOfRangeException(nameof(value),
34                  value, $"{nameof(Quantity)} must be >= 0");
35            }
36
37            quantity = value;
38         }
39      }
40
```

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

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

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

```
59
60      // return string representation of Invoice object
61      public override string ToString() =>
62         $"invoice:\npart number: {PartNumber} ({PartDescription})\n" +
63         $"quantity: {Quantity}\nprice per item: {PricePerItem:C}";
64
65      // method required to carry out contract with interface IPayable
66      public decimal GetPaymentAmount() => Quantity * PricePerItem;
67   }
```

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

Software Engineering Observation 12.6

C# does not allow derived classes to inherit from more than one base class, but it does allow a class to inherit from a base class and implement any number of interfaces.

12.7.3 Creating Class Invoice (Cont.)

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

12.7.4 Modifying Class Employee to Implement Interface IPayable

- Figure 12.13 contains the `Employee` class, modified to implement interface `IPayable`.
- Notice that `GetPaymentAmount` simply calls `Employee`'s `abstract` method `Earnings`.
- At execution time, when `GetPaymentAmount` is called on an object of an `Employee` derived class, `GetPaymentAmount` calls that class's concrete `Earnings` method, which knows how to calculate earnings for objects of that derived-class type.

```
1  // Fig. 12.13: Employee.cs
2  // Employee abstract base class that implements interface IPayable.
3  public abstract class Employee : IPayable
4  {
5     public string FirstName { get; }
6     public string LastName { get; }
7     public string SocialSecurityNumber { get; }
8
9     // three-parameter constructor
10    public Employee(string firstName, string lastName,
11       string socialSecurityNumber)
12    {
13       FirstName = firstName;
14       LastName = lastName;
15       SocialSecurityNumber = socialSecurityNumber;
16    }
17
```

Fig. 12.13 | `Employee` abstract base class that implements interface `IPayable`. (Part 1 of 2.)

```
18      // return string representation of Employee object, using properties
19      public override string ToString() => $"{FirstName} {LastName}\n" +
20         $"social security number: {SocialSecurityNumber}";
21
22      // abstract method overridden by derived classes
23      public abstract decimal Earnings(); // no implementation here
24
25      // implementing GetPaymentAmount here enables the entire Employee
26      // class hierarchy to be used in an app that processes IPayables
27      public decimal GetPaymentAmount() => Earnings();
28  }
```

Fig. 12.13 | `Employee` abstract base class that implements interface `IPayable`. (Part 2 of 2.)

12.7.5 Modifying Class SalariedEmployee for Use with IPayable

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

Software Engineering Observation 12.7

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

Software Engineering Observation 12.8

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

12.7.6 Using Interface IPayable to Process Invoices and Employees Polymorphically

- `PayableInterfaceTest` (Fig. 12.14) illustrates that interface `IPayable` can be used to processes a set of `Invoices` and `Employees` polymorphically in a single app.

Software Engineering Observation 12.9

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

```
1  // Fig. 12.14: PayableInterfaceTest.cs
2  // Tests interface IPayable with disparate classes.
3  using System;
4  using System.Collections.Generic;
5
6  class PayableInterfaceTest
7  {
8     static void Main()
9     {
10       // create a List<IPayable> and initialize it with four
11       // objects of classes that implement interface IPayable
12       var payableObjects = new List<IPayable>() {
13          new Invoice("01234", "seat", 2, 375.00M),
14          new Invoice("56789", "tire", 4, 79.95M),
15          new SalariedEmployee("John", "Smith", "111-11-1111", 800.00M),
16          new SalariedEmployee("Lisa", "Barnes", "888-88-8888", 1200.00M)};
17
```

Fig. 12.14 | Tests interface `IPayable` with disparate classes. (Part 1 of 3.)

```
18          Console.WriteLine(
19             "Invoices and Employees processed polymorphically:\n");
20
21          // generically process each element in payableObjects
22          foreach (var payable in payableObjects)
23          {
24             // output payable and its appropriate payment amount
25             Console.WriteLine($"{payable}");
26             Console.WriteLine(
27                $"payment due: {payable.GetPaymentAmount():C}\n");
28          }
29       }
30    }
```

Fig. 12.14 | Tests interface `IPayable` with disparate classes. (Part 2 of 3.)

```
Invoices and Employees processed polymorphically:

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

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

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

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

Fig. 12.14 | Tests interface `IPayable` with disparate classes. (Part 3 of 3.)

12.7.7 Common Interfaces of the .NET Framework Class Library

- Figure 12.15 overviews several commonly used Framework Class Library interfaces.

Interface	Description
`IComparable`	C# contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare simple-type values. Section 10.13 showed that you can overload these operators for your own types. Interface `IComparable` can be used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, `CompareTo`, which compares the object that calls the method to the object passed as an argument. Classes must implement `CompareTo` to return a value indicating whether the object on which it's invoked is less than (negative integer return value), equal to (0 return value) or greater than (positive integer return value) the object passed as an argument, using any criteria you specify. For example, if class `Employee` implements `IComparable`, its `CompareTo` method could compare `Employee` objects by their earnings amounts. Interface `IComparable` is commonly used for ordering objects in a collection such as an array. We use `IComparable` in Chapter 20, Generics, and Chapter 21, Generic Collections; Functional Programming with LINQ/PLINQ.

Fig. 12.15 | Common interfaces of the .NET Framework Class Library. (Part 1 of 4.)

Interface	Description
`IComponent`	Implemented by any class that represents a component, including Graphical User Interface (GUI) controls (such as buttons or labels). Interface `IComponent` defines the behaviors that components must implement. We discuss `IComponent` and many GUI controls that implement this interface in Chapter 14, Graphical User Interfaces with Windows Forms: Part 1, and Chapter 15, Graphical User Interfaces with Windows Forms: Part 2.

Fig. 12.15 | Common interfaces of the .NET Framework Class Library. (Part 2 of 4.)

Interface	Description
`IDisposable`	Implemented by classes that must provide an explicit mechanism for *releasing* resources. Some resources can be used by only one program at a time. In addition, some resources, such as files on disk, are unmanaged resources that, unlike memory, cannot be released by the garbage collector. Classes that implement interface `IDisposable` provide a `Dispose` method that can be called to explicitly release resources that are explicitly associated with an object. We discuss `IDisposable` briefly in Chapter 13, Exception Handling: A Deeper Look. You can learn more about this interface at `http://msdn.microsoft.com/library/system.idisposable`. The MSDN article *Implementing a Dispose Method* at `http://msdn.microsoft.com/library/fs2xkftw` discusses the proper implementation of this interface in your classes.

Fig. 12.15 | Common interfaces of the .NET Framework Class Library. (Part 3 of 4.)

Interface	Description
`IEnumerator`	Used for iterating through the elements of a *collection* (such as an array or a `List`) one element at a time—the `foreach` statement uses an `IEnumerator` object to iterate through elements. Interface `IEnumerator` contains method `MoveNext` to move to the next element in a collection, method `Reset` to move to the position before the first element and property `Current` to return the object at the current location. We use `IEnumerator` in Chapter 21. All `IEnumberable` objects (Chapter 9) provide a `GetEnumerator` method that returns an `IEnumerator` object.

Fig. 12.15 | Common interfaces of the .NET Framework Class Library. (Part 4 of 4.)

Exercises

- 12.7, 12.8, and 12.11 (pp. 505-506 of the textbook)