Object-Oriented Programming: Polymorphism
One Ring to rule them all,
One Ring to find them,
One Ring to bring them all
and in the darkness bind them.

– J. R. R. Tolkien

General propositions do not
decide concrete cases.

– Oliver Wendell Holmes
A philosopher of imposing stature doesn’t think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.

– Alfred North Whitehead

Why art thou cast down, O my soul?

– Psalms 42:5
OBJECTIVES

In this chapter you will learn:

- What polymorphism is.
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
OBJECTIVES

- How polymorphism makes systems extensible and maintainable.
- To determine an object’s type at execution time.
- To declare and implement interfaces.
12.1 Introduction
12.2 Polymorphic Video Game
12.3 Demonstrating Polymorphic Behavior
12.4 Abstract Classes and Methods
12.5 Case Study: Payroll System Class Hierarchy Using Polymorphism
12.6 NotOverridable Methods and NotInheritable Classes
12.7 Case Study: Creating and Using Interfaces
12.8 (Optional) Software Engineering Case Study: Incorporating Inheritance and Polymorphism into the ATM System
12.1 Introduction

• Polymorphism enables us to “program in the general.”
• New classes can be added to an inheritance hierarchy with little or no modification to the general portions of the program.
12.2 Polymorphic Video Game

• Suppose we design a video game with objects of many types, inheriting from class `SpaceObject`.
• The screen manager makes the same method call, `Draw`, for each object.
• Each derived class implements `Draw` in a manner appropriate to that class.
Software Engineering Observation 12.1

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.
• To perform a derived class-specific operation, the program must **downcast** a base class reference.

• Figure 12.1 shows three ways to use base class and derived class variables.

```vbnet
Module PolymorphismTest
    Sub Main()
        Dim commissionEmployee1 As CommissionEmployee
        Dim basePlusCommissionEmployee As BasePlusCommissionEmployee

        ' assign base class reference to base class variable
        commissionEmployee1 = New CommissionEmployee("Sue", "Jones", "222-22-2222", 10000D, 0.06)

        ' assign derived class reference to derived class variable
        basePlusCommissionEmployee = New BasePlusCommissionEmployee("Bob", "Lewis", "333-33-3333", 5000D, 0.04, 300D)

        ' invoke ToString on base class object using base class variable
        Console.WriteLine("Call CommissionEmployee's ToString with " & _
            "base class reference to base class object: " & vbNewLine & _
            commissionEmployee1.ToString() & vbNewLine)
    End Sub
End Module
```

**Fig. 12.1** | Assigning base class and derived class references to base class and derived class variables. (Part 1 of 3.)
' invoke ToString on derived class object using derived class variable
Console.WriteLine("Call BasePlusCommissionEmployee's ToString " & _
    "with derived class reference to derived class object: " & _

' assign derived class reference to base class variable
Dim commissionEmployee2 As CommissionEmployee = _
    basePlusCommissionEmployee

' invoke ToString on derived class object using base class variable
Console.WriteLine("Call BasePlusCommissionEmployee's ToString " & _
    "with base class reference to derived class object: " & _
    vbNewLine & commissionEmployee2.ToString())
End Sub ' Main
End Module ' PolymorphismTest

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

Call BasePlusCommissionEmployee's ToString with derived class reference to derived class object:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5,000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee's ToString with base class reference to derived class object:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5,000.00
commission rate: 0.04
base salary: 300.00

**Fig. 12.1** Assigning base class and derived class references to base class and derived class variables. (Part 3 of 3.)
12.4 Abstract Classes and Methods

- **Abstract classes** are incomplete classes which cannot be instantiated.
- In the *Shape* hierarchy, we could derive concrete classes from abstract class *TwoDimensionalShape*.
- A programmer can use an abstract base class as a parameter type.
- These methods can be passed an object of any concrete class that inherits the base class.
• Make a class abstract by declaring it with `MustInherit`.

• **Abstract methods** have keyword `MustOverride` in their declarations:

```vbnet
Public MustOverride Sub Draw() ' abstract method
```

• A class that contains any abstract methods must be declared as an abstract class.
12.4 Abstract Classes and Methods (Cont.)

Software Engineering Observation 12.2

An abstract class typically contains one or more abstract methods that derived classes must override if the derived classes are to be concrete. The instance variables and concrete methods of an abstract class are subject to the normal rules of inheritance.
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 in a derived class is a compilation error unless the derived class is also declared MustInherit.
12.4 Abstract Classes and Methods (Cont.)

• It is common to declare an *iterator class* that can represent all the objects in a collection, such as an array or a *List*.

• An *ArrayList* of objects of class *TwoDimensionalShape* could contain objects from derived classes *Square*, *Circle*, *Triangle* and so on.
12.5 Case Study: Payroll System Class Hierarchy Using Polymorphism

• We create an enhanced employee hierarchy to solve the following problem:

A company wants to implement an application that performs its payroll calculations on a weekly basis. Salaried employees are paid a fixed weekly salary, hourly employees are paid by the hour and receive overtime pay (1.5 times the regular hourly salary), commission employees are paid a percentage of their sales, and other employees receive a base salary plus a percentage of their sales.
• The UML class diagram in Fig. 12.2 shows an inheritance hierarchy.

• Note that abstract class name Employee is italicized.

Fig. 12.2 | Employee hierarchy UML class diagram.
12.5 Case Study: Payroll System Class Hierarchy Using Polymorphism (Cont.)

12.5.1 Creating Abstract Base Class Employee

- Class Employee will provide methods CalculateEarnings and ToString, and properties that manipulate instance variables.
- We declare CalculateEarnings as MustOverride so that each derived class provides an appropriate implementation.
- Each derived class of Employee overrides method ToString to include the employee’s type.
12.5 Case Study: Payroll System Class Hierarchy Using Polymorphism (Cont.)

- The diagram in Fig. 12.3 shows polymorphic variations of the Employee class’s methods.

<table>
<thead>
<tr>
<th>Class</th>
<th>CalculateEarnings</th>
<th>ToString</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>MustOverride</td>
<td>FirstName LastName social security number: SSN</td>
</tr>
<tr>
<td>Salaried-Employee</td>
<td>WeeklySalary</td>
<td>salaried employee: firstName lastName social security number: SSN weekly salary: WeeklySalary</td>
</tr>
<tr>
<td>Hourly-Employee</td>
<td>If Hours &lt;= 40</td>
<td>hourly employee: firstName lastName social security number: SSN hourly wage: Wage; hours worked: Hours</td>
</tr>
<tr>
<td></td>
<td>Wage = Hours</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If Hours &gt; 40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>40 * Wage + ((Hours - 40) * Wage + 1.5)</td>
<td></td>
</tr>
<tr>
<td>Commission-Employee</td>
<td>CommissionRate * GrossSales</td>
<td>commission employee: firstName lastName social security number: SSN gross sales: GrossSales; commission rate: CommissionRate</td>
</tr>
<tr>
<td>BasePlus-Commission-Employee</td>
<td>(CommissionRate * GrossSales) + BaseSalary</td>
<td>base-salaried commission employee: firstName lastName social security number: SSN gross sales: GrossSales; commission rate: CommissionRate; base salary: BaseSalary</td>
</tr>
</tbody>
</table>

**Fig. 12.3** | Polymorphic interface for the Employee hierarchy classes.
Class Employee is shown in Fig. 12.4.

```vbnet
Public MustInherit Class Employee
    Private firstNameValue As String
    Private lastNameValue As String
    Private socialSecurityNumberValue As String

    ' three-argument constructor
    Public Sub New(ByVal first As String, ByVal last As String, ByVal ssn As String)
        FirstName = first
        LastName = last
        SocialSecurityNumber = ssn
    End Sub ' New

    ' property FirstName
    Public Property FirstName() As String
        Get
            Return firstNameValue
        End Get
    END Property
```

Fig. 12.4 Employee abstract base class. (Part 1 of 3.)
```vbnet
Set(ByVal first As String)
    firstNameValue = first
End Set
End Property ' FirstName

' property LastName
Public Property LastName() As String
    Get
        Return lastNameValue
    End Get
    Set(ByVal last As String)
        lastNameValue = last
    End Set
End Property ' LastName
```

Fig. 12.4 | Employee abstract base class. (Part 2 of 3.)
' property SocialSecurityNumber
Public Property SocialSecurityNumber() As String
    Get
        Return socialSecurityNumberValue
    End Get

    Set(ByVal ssn As String)
        socialSecurityNumberValue = ssn
    End Set
End Property ' SocialSecurityNumber

' return String representation of Employee object
Public Overrides Function ToString() As String
    Return (String.Format("{0} {1}", FirstName, LastName) & 
            vbCrLf & "social security number: " & SocialSecurityNumber)
End Function ' ToString

' abstract method overridden by derived class
Public MustOverride Function CalculateEarnings() As Decimal
End Class ' Employee

Fig. 12.4 | Employee abstract base class. (Part 3 of 3.)
Class SalariedEmployee (Fig. 12.5) is a concrete class which inherits an abstract class.

```vbnet
Public Class SalariedEmployee
    Inherits Employee

    Private weeklySalaryValue As Decimal ' employee's weekly salary

    ' four-argument constructor
    Public Sub New(ByVal first As String, ByVal last As String, ByVal ssn As String, ByVal salary As Decimal)
        MyBase.New(first, last, ssn) ' pass to Employee constructor
        WeeklySalary = salary ' validate and store salary
    End Sub ' New

    ' property WeeklySalary
    Public Property WeeklySalary() As Decimal
        Get
            Return weeklySalaryValue
        End Get
    End Property

End Class
```

Fig. 12.5 | SalariedEmployee class derived from class Employee. (Part 1 of 2.)
```vbnet
Set(ByVal salary As Decimal)
    If salary < 0D Then ' validate salary
        weeklySalaryValue = 0D
    Else
        weeklySalaryValue = salary
    End If
End Set
End Property ' WeeklySalary

' calculate earnings; override abstract method CalculateEarnings
Public Overrides Function CalculateEarnings() As Decimal
    Return WeeklySalary
End Function ' CalculateEarnings

' return String representation of SalariedEmployee object
Public Overrides Function ToString() As String
    Return ("salaried employee: " & MyBase.ToString() & vbNewLine & _
            String.Format("weekly salary {0:C}", WeeklySalary))
End Function ' ToString
End Class ' SalariedEmployee
```

Fig. 12.5 | SalariedEmployee class derived from class Employee. (Part 2 of 2.)
• Class **HourlyEmployee** (Fig. 12.6) also inherits class **Employee**.

```vbnet
' Fig. 12.6: HourlyEmployee.vb
' HourlyEmployee class inherits Employee.
Public Class HourlyEmployee
    Inherits Employee

    Private wageValue As Decimal ' wage per hour
    Private hoursValue As Decimal ' hours worked for week

    ' five-argument constructor
    Public Sub New(ByVal first As String, ByVal last As String, _
                    ByVal ssn As String, ByVal hourlyWage As Decimal, _
                    ByVal hoursWorked As Decimal)
        MyBase.New(first, last, ssn) ' pass to Employee constructor
        Wage = hourlyWage ' validate and store hourly wage
        Hours = hoursWorked ' validate and store hours worked
    End Sub ' New
```

**Fig. 12.6** | HourlyEmployee class derived from class Employee. (Part 1 of 4.)
' property Wage
Public Property Wage() As Decimal
    Get
        Return wageValue
    End Get

    Set(ByVal hourlyWage As Decimal)
        If hourlyWage < 0D Then ' validate hourly wage
            wageValue = 0D
        Else
            wageValue = hourlyWage
        End If
    End Set
End Property ' Wage

' property Hours
Public Property Hours() As Decimal
    Get
        Return hoursValue
    End Get
Set(ByVal hoursWorked As Decimal)
    If (hoursWorked >= 0.0) AndAlso (hoursWorked <= 168.0) Then
        hoursValue = hoursWorked ' valid weekly hours
    Else
        hoursValue = 0
    End If
End Set

End Property ' Hours

' calculate earnings; override abstract method CalculateEarnings
Public Overrides Function CalculateEarnings() As Decimal
    If Hours <= 40 Then ' no overtime
        Return Wage * Hours
    Else
        Return 40 * Wage + ((Hours - 40) * Wage * 1.5D)
    End If
End Function ' CalculateEarnings

Fig. 12.6 | HourlyEmployee class derived from class Employee. (Part 3 of 4.)
' return String representation of HourlyEmployee object
Public Overrides Function ToString() As String
    Return ("hourly employee: " & MyBase.ToString() & vbCrLf & _
        String.Format("hourly wage: {0:C}; hours worked: {1}", _
        Wage, Hours))
End Function ' ToString
End Class ' HourlyEmployee

Outputting the employee’s type, information produced by Employee’s ToString method and more specific information.

Fig. 12.6 | HourlyEmployee class derived from class Employee. (Part 4 of 4.)
Class **CommissionEmployee** (Fig. 12.7) is another extension of **Employee**.

```vbnet
' Fig. 12.7: CommissionEmployee.vb
' CommissionEmployee class inherits Employee.
Public Class CommissionEmployee
    Inherits Employee

    Private grossSalesValue As Decimal
    Private commissionRateValue As Double

    ' five-argument constructor
    Public Sub New(ByVal first As String, ByVal last As String, ByVal ssn As String, ByVal sales As Decimal, ByVal rate As Double)
        MyBase.New(first, last, ssn)
        GrossSales = sales ' validate and store gross sales
        CommissionRate = rate ' validate and store commission rate
    End Sub ' New

    ' property GrossSales
    Public Property GrossSales() As Decimal
        Get
            Return grossSalesValue
        End Get
    End Property

Fig. 12.7 | CommissionEmployee class derived from Employee. (Part 1 of 3.)
```
```vbnet
    Set(ByVal sales As Decimal)
        If sales < 0 Then ' validate gross sales
            grossSalesValue = 0
        Else
            grossSalesValue = sales
        End If
    End Set
End Property ' GrossSales

' property CommissionRate
Public Property CommissionRate() As Double
    Get
        Return commissionRateValue
    End Get
```

**Fig. 12.7** | CommissionEmployee class derived from Employee. (Part 2 of 3.)
CommissionEmployee.vb

(3 of 3)

Fig. 12.7 | CommissionEmployee class derived from Employee. (Part 3 of 3.)
BasePlusCommissionEmployee (Fig. 12.8) inherits CommissionEmployee and therefore is an indirect derived class of Employee.

```vbnet
' Fig. 12.8: BasePlusCommissionEmployee.vb
' BasePlusCommissionEmployee class inherits CommissionEmployee.
Public Class BasePlusCommissionEmployee
    Inherits CommissionEmployee

    Private baseSalaryValue As Decimal ' base salary

    ' six-argument constructor
    Public Sub New(ByVal first As String, ByVal last As String, ByVal ssn As String, ByVal sales As Decimal, ByVal rate As Decimal, ByVal salary As Decimal)
        MyBase.New(first, last, ssn, sales, rate)
        BaseSalary = salary
    End Sub ' New

    ' property BaseSalary
    Public Property BaseSalary() As Decimal
        Get
            Return baseSalaryValue
        End Get
    End Property

Fig. 12.8 | BasePlusCommissionEmployee derived from CommissionEmployee. (Part 1 of 2.)
```
```vbnet
Set(ByVal salary As Decimal)
    If salary < 0 Then ' validate salary
        baseSalaryValue = 0
    Else
        baseSalaryValue = salary
    End If
End Set ' BaseSalary

' calculate earnings; override method CalculateEarnings
Public Overrides Function CalculateEarnings() As Decimal
    Return BaseSalary + MyBase.CalculateEarnings()
End Function ' CalculateEarnings

' return String representation of BasePlusCommissionEmployee object
Public Overrides Function ToString() As String
    Return ("base-salaried " & MyBase.ToString() & 
        String.Format("; base salary: {0:C}", BaseSalary))
End Function ' ToString
End Class ' BasePlusCommissionEmployee
```

**Fig. 12.8** | BasePlusCommissionEmployee derived from CommissionEmployee. (Part 2 of 2.)
• The program in Fig. 12.9 creates an object of each of the four concrete classes.

```vbnet
Module PayrollSystemTest
    Sub Main()
        ' create derived class objects
        Dim salariedEmployee As New SalariedEmployee("John", "Smith", "111-11-1111", 800)
        Dim hourlyEmployee As New HourlyEmployee("Karen", "Price", "222-22-2222", 16.75D, 40)
        Dim commissionEmployee As New CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, 0.06)
        Dim basePlusCommissionEmployee As New BasePlusCommissionEmployee("Bob", "Lewis", "444-44-4444", 5000, 0.04, 300)

        ' display each employee's info non-polymorphically
        Console.WriteLine("Employees processed individually:" & vbCrLf)
        Console.WriteLine(salariedEmployee.ToString() & vbCrLf & String.Format("earned: {0:C}", salariedEmployee.CalculateEarnings())) & vbCrLf)
    End Sub
End Module
```

**Fig. 12.9** | Employee class hierarchy test program. (Part 1 of 6.)
Console.WriteLine(hourlyEmployee.ToString() & vbCrLf & 
    String.Format("earned: {0:C}", _
    hourlyEmployee.CalculateEarnings())) & vbCrLf)  
Console.WriteLine(commissionEmployee.ToString() & vbCrLf & 
    String.Format("earned: {0:C}", _
    commissionEmployee.CalculateEarnings())) & vbCrLf)  
Console.WriteLine(basePlusCommissionEmployee.ToString() & 
    vbCrLf & String.Format("earned: {0:C}", _
    basePlusCommissionEmployee.CalculateEarnings())) & vbCrLf)

' create four-element Employee array
Dim employees() As Employee = {salariedEmployee, hourlyEmployee, _
    commissionEmployee, basePlusCommissionEmployee}

Console.WriteLine("Employees processed polymorphically:" & 
    vbCrLf)
' polymorphically process each element in array employees
For Each currentEmployee In employees
    Console.WriteLine(currentEmployee.ToString())

' determine if currentEmployee is a BasePlusCommissionEmployee
If (TypeOf currentEmployee Is BasePlusCommissionEmployee) Then
    ' downcast Employee reference to BasePlusCommissionEmployee
    Dim employee As BasePlusCommissionEmployee = _
    TryCast(currentEmployee, BasePlusCommissionEmployee)
    employee.BaseSalary *= 1.1D
    Console.WriteLine(String.Format(_
        "new base salary with 10% increase is: {0:C}", _
        employee.BaseSalary))
End If

Fig. 12.9 | Employee class hierarchy test program. (Part 3 of 6.)
Outputting Employee object types polymorphically.
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
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: $600.00

(continued on next page...)
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
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 PayrollSystem.SalariedEmployee
Employee 1 is a PayrollSystem.HourlyEmployee
Employee 2 is a PayrollSystem.CommissionEmployee
Employee 3 is a PayrollSystem.BasePlusCommissionEmployee

Fig. 12.9 | Employee class hierarchy test program. (Part 6 of 6.)
12.5 Case Study: Payroll System Class Hierarchy Using Polymorphism (Cont.)

• The output polymorphically displays the earnings and types of different Employee objects.

Using Expression `typeof ... is` to Determine Object Type

• `typeof ... is` tests for a particular type.
12.5 Case Study: Payroll System Class Hierarchy Using Polymorphism (Cont.)

Using `TryCast` to Downcast from a Base Class to a Derived Class Type

- `TryCast` attempts to downcast a base class type to a derived class type.
- If the object is not an instance of the derived class, `TryCast` returns `Nothing`.
- `GetType` returns a `Type` object which contains information about an object’s type.
Common Programming Error 12.3
Assigning a base class variable to a derived class variable (without an explicit cast) is a compilation error.

Error-Prevention Tip 12.1
Before performing a downcast, use the Typeof...Is expression to ensure that the object is indeed an object of an appropriate derived class type. Then use DirectCast to downcast from the base class type to the derived class type.
12.6 NotOverridable Methods and NotInheritable Classes

• A method that is declared NotOverridable cannot be overridden.

• An inherited method can be declared NotOverridable in a derived class to prevent further inheritance.

• A class that is declared NotInheritable cannot be a base class.

Common Programming Error 12.4

Attempting to inherit a NotInheritable class is a compilation error.
12.7 Case Study: Creating and Using Interfaces

• A Visual Basic interface describes a set of methods that can be called on an object.

• An interface declaration begins with keyword Interface and can contain abstract methods and properties.

• To use an interface, a concrete class must specify that it Implements the interface and must implement each method.
12.7 Case Study: Creating and Using Interfaces (Cont.)

Common Programming Error 12.5

It is a compilation error to explicitly declare interface methods Public.

Common Programming Error 12.6

In Visual Basic, an Interface should be declared only as Public or Friend; the declaration of an Interface as Private or Protected is only possible within another type.
Common Programming Error 12.7

Failing to implement any method of an interface in a concrete class that implements the interface results in a syntax error indicating that the class must be declared MustInherit.
12.7 Case Study: Creating and Using Interfaces (Cont.)

12.7.1 Developing an IPayable Hierarchy

• Interface IPayable contains method GetPaymentAmount, which returns a Decimal, and ToString.

• Classes Invoice and Employee will implement interface IPayable.

• A program will be able to invoke GetPaymentAmount on Invoices, Employees and any future objects.
12.7 Case Study: Creating and Using Interfaces (Cont.)

Good Programming Practice 12.1

When declaring a method in an interface, choose a method name that describes the method’s purpose in a general manner, because the method may be implemented by many unrelated classes.
12.7 Case Study: Creating and Using Interfaces (Cont.)

• The UML class diagram in Fig. 12.10 shows the class and interface hierarchy.

• The UML expresses the interface implementation through an association known as a realization.

Fig. 12.10 | IPayable interface hierarchy UML class diagram.
• Interface `IPayable` (Fig. 12.11) contains abstract method `GetPaymentAmount`.

```vbnet
1 ' Fig. 12.11: IPayable.vb
2 ' IPayable interface declaration.
3 Public Interface IPayable
4 Function GetPaymentAmount() As Decimal ' calculate payment
5 End Interface ' IPayable
```

**Fig. 12.11** | `IPayable` interface declaration.
• **Invoice** (Fig. 12.12) represents a simple invoice that contains billing information for only one kind of part.

```vbnet
' Fig. 12.12: Invoice.vb
' Invoice class implements IPayable.
Public Class Invoice
  Implements IPayable
  Private partNumberValue As String
  Private partDescriptionValue As String
  Private quantityValue As Integer
  Private pricePerItemValue As Decimal

  ' four-argument constructor
  Public Sub New(ByVal part As String, ByVal description As String, _
                  ByVal count As Integer, ByVal price As Decimal)
    PartNumber = part
    PartDescription = description
    Quantity = count ' validate quantity
    PricePerItem = price ' validate price per item
  End Sub ' New
```

**Fig. 12.12** Invoice class that implements interface IPayable. (Part 1 of 5.)
Outline

Invoice.vb

(2 of 5)

Fig. 12.12 | Invoice class that implements interface IPayable. (Part 2 of 5.)
' property Quantity
Public Property Quantity() As Integer
    Get
        Return quantityValue
    End Get
    Set(ByVal count As Integer)
        If count < 0 Then ' validate quantity
            quantityValue = 0
        Else
            quantityValue = count
        End If
    End Set
End Property ' Quantity

' property PricePerItem
Public Property PricePerItem() As Decimal
    Get
        Return pricePerItemValue
    End Get

Fig. 12.12 | Invoice class that implements interface IPayable. (Part 3 of 5.)
Set(ByVal price As Decimal)
    If price < 0 Then ' validate price
        pricePerItemValue = 0
    Else
        pricePerItemValue = price
    End If
End Set
End Property ' PricePerItem

' function required to carry out contract with interface IPayable
Public Function GetPaymentAmount() As Decimal Implements IPayable.GetPaymentAmount
    Return Quantity * PricePerItem ' calculate total cost
End Function ' GetPaymentAmount

Fig. 12.12 | Invoice class that implements interface IPayable. (Part 4 of 5.)
Fig. 12.12 | Invoice class that implements interface IPayable. (Part 5 of 5.)
Figure 12.13 contains the modified **Employee** class.

```vbnet
' Fig. 12.13: Employee.vb
' Employee abstract base class implements IPayable.
Public MustInherit Class Employee
  Implements IPayable
  
  Private firstNameValue As String
  Private lastNameValue As String
  Private socialSecurityNumberValue As String

  ' three-argument constructor
  Public Sub New(ByVal first As String, ByVal last As String, ByVal ssn As String)
    FirstName = first
    LastName = last
    SocialSecurityNumber = ssn
  End Sub
End Class
```

**Employee.vb**

---

© 2009 Pearson Education, Inc. All rights reserved.
' property FirstName
Public Property FirstName() As String
    Get
        Return firstNameValue
    End Get
    Set(ByVal first As String)
        firstNameValue = first
    End Set
End Property ' FirstName

' property LastName
Public Property LastName() As String
    Get
        Return lastNameValue
    End Get
    Set(ByVal last As String)
        lastNameValue = last
    End Set
End Property ' LastName

Fig. 12.13 | Employee class that implements interface IPayable. (Part 2 of 3.)
' property SocialSecurityNumber
Public Property SocialSecurityNumber() As String
    Get
        Return socialSecurityNumberValue
    End Get

    Set(ByVal ssn As String)
        socialSecurityNumberValue = ssn
    End Set
End Property ' SocialSecurityNumber

' return String representation of Employee object
Public Overrides Function ToString() As String
    Return (String.Format("{0} {1}", FirstName, LastName) & _
        vbCrLf & "social security number: " & SocialSecurityNumber)
End Function ' ToString

' Note: We do not implement IPayable function GetPaymentAmount here
Public MustOverride Function GetPaymentAmount() As Decimal _
    Implements IPayable.GetPaymentAmount
End Class ' Employee
Figure 12.14 contains a modified version of SalariedEmployee.

```vbnet
' Fig. 12.14: SalariedEmployee.vb
' SalariedEmployee class inherits Employee, which implements IPayable.
Public Class SalariedEmployee
    Inherits Employee

    Private weeklySalaryValue As Decimal

    ' four-argument constructor
    Public Sub New(ByVal first As String, ByVal last As String, ByVal ssn As String, ByVal salary As Decimal)
        MyBase.New(first, last, ssn) ' pass to Employee constructor
        WeeklySalary = salary ' validate and store salary
    End Sub ' New

    ' property WeeklySalary
    Public Property WeeklySalary() As Decimal
        Get
            Return weeklySalaryValue
        End Get
    End Property

Fig. 12.14 | SalariedEmployee class that implements interface IPayable method GetPaymentAmount. (Part 1 of 2.)
```
Set(ByVal salary As Decimal)
  If salary < 0D Then ' validate salary
    weeklySalaryValue = 0D
  Else
    weeklySalaryValue = salary
  End If
End Set
End Property ' WeeklySalary

' calculate earnings; implements interface IPayable method
' GetPaymentAmount that was MustOverride in base class Employee
Public Overrides Function GetPaymentAmount() As Decimal
  Return WeeklySalary
End Function ' GetPaymentAmount

' return String representation of SalariedEmployee object
Public Overrides Function ToString() As String
  Return ("salaried employee: " & MyBase.ToString() & vbCrLf & _
          String.Format("weekly salary {0:C}", WeeklySalary))
End Function ' ToString
End Class ' SalariedEmployee

Fig. 12.14 | SalariedEmployee class that implements interface IPayable
method GetPaymentAmount. (Part 2 of 2.)
Software Engineering Observation 12.3

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 12.4

If a parameter has a base class type, it can receive either a base class or derived class reference as an argument. If a parameter has an interface type, it can receive a reference to an object of any class that implements the interface.
PayableInterfaceTest (Fig. 12.15) illustrates that interface IPayable can be used to process a set of Invoices and Employees polymorphically.

```
' Fig. 12.15: PayableInterfaceTest.vb
' Testing interface IPayable.
Module PayableInterfaceTest
    Sub Main()
        ' create four-element IPayable array
        Dim payableObjects() As IPayable = New IPayable(3) {}
        ' populate array with objects that implement IPayable
        payableObjects(0) = New Invoice("01234", "seat", 2, 375D)
        payableObjects(1) = New SalariedEmployee(_
            "John", "Smith", "111-11-1111", 800D)
        payableObjects(2) = New Invoice("56789", "tire", 4, 79.95D)
        payableObjects(3) = New SalariedEmployee(_
            "Lisa", "Barnes", "888-88-8888", 1200D)
        Console.WriteLine(_
            "Invoices and Employees processed polymorphically:" & vbNewLine)
    End Sub
End Module
```

Fig. 12.15 | IPayable interface test program processing Invoices and Employees polymorphically. (Part 1 of 3.)
Generically process each element in array `payableObjects`

```vbnet
For Each currentPayable In payableObjects
    ' output currentPayable and its appropriate payment amount
    Console.WriteLine(currentPayable.ToString() & vbCrLf & String.Format("payment due: {0:C}",
    currentPayable.GetPaymentAmount()) & vbCrLf)
Next
End Sub ' Main
End Module ' PayableInterfaceTest
```

Fig. 12.15 | `IPayable` interface test program processing Invoices and Employees polymorphically. (Part 2 of 3.)
Software Engineering Observation 12.5

Using a base class reference, you can polymorphically invoke any method specified in the base class declaration. Using an interface reference, you can polymorphically invoke any method specified in the interface declaration.
12.7 Case Study: Creating and Using Interfaces (Cont.)

12.7.7 Common Interfaces of the .NET Framework Class Library

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IComparable</td>
<td>IComparable is used to allow objects to be compared to one another.</td>
</tr>
<tr>
<td>IComponent</td>
<td>Implemented by any class that represents a component, including GUI controls.</td>
</tr>
<tr>
<td>IEnumerable</td>
<td>Contains a GetEnumerator method, which returns an enumerator for the collection.</td>
</tr>
<tr>
<td>IEnumerator</td>
<td>Used for iterating through the elements of a collection one element at a time.</td>
</tr>
</tbody>
</table>

**Fig. 12.16** | Common interfaces of the .NET Class Library.
12.8 Incorporating Inheritance and Polymorphism into the ATM System

• Fig. 12.17 shows the attributes and operations of the three transaction classes.

**Fig. 12.17** | Attributes and operations of classes BalanceInquiry, Withdrawal and Deposit.
12.8 Incorporating Inheritance and Polymorphism into the ATM System (Cont.)

- The UML specifies a relationship called a **generalization** to model inheritance.

- Figure 12.18 models the inheritance relationship between **Transaction** and its three derived classes.

![Class diagram modeling the generalization relationship between Transaction and its derived classes.](image_url)

**Fig. 12.18** | Class diagram modeling the generalization (i.e., inheritance) relationship between the base class Transaction and its derived classes BalanceInquiry, Withdrawal, and Deposit.
12.8 Incorporating Inheritance and Polymorphism into the ATM System (Cont.)

• Include operation \textbf{Execute} in base class \textit{Transaction} so that the ATM can invoke each derived class’s overridden version.

• Figure 12.19 presents an updated class diagram.
12.8 Incorporating Inheritance and Polymorphism into the ATM System (Cont.)

Fig. 12.19 | Class diagram of the ATM system (incorporating inheritance). Note that MustInherit class name Transaction appears in italics.

© 2009 Pearson Education, Inc. All rights reserved.
Software Engineering Observation 12.6

A complete class diagram shows all the associations among classes, and all the attributes and operations for each class. When the number of class attributes, operations and associations is substantial, a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and operations.
12.8 Incorporating Inheritance and Polymorphism into the ATM System (Cont.)

Fig. 12.20| Class diagram after incorporating inheritance into the system.
1. **Transaction** is a generalization of class **Withdrawal** (Fig. 12.21).

```
' Class Withdrawal represents an ATM withdrawal transaction.
Public Class Withdrawal
    Inherits Transaction
End Class ' Withdrawal
```

**Fig. 12.21** | Visual Basic code for shell of class withdrawal.
2. **Withdrawal** must implement the abstract operations of its base class (Fig. 12.22).

```vbnet
' Class Withdrawal represents an ATM withdrawal transaction.
Public Class Withdrawal
    Inherits Transaction

    ' attributes
    Private amount As Decimal ' amount to withdraw
    Private keypad As Keypad ' reference to keypad
    Private cashDispenser As CashDispenser ' reference to cash dispenser

    ' parameterless constructor
    Public Sub New()
        ' constructor body code
    End Sub ' New

    ' method that overrides Execute
    Public Overrides Sub Execute()
        ' Execute method body code
    End Sub ' Execute

End Class ' Withdrawal
```

**Fig. 12.22** | Visual Basic code for class **Withdrawal** based on Fig. 12.19 and Fig. 12.20.