Polymorphism

OBJECTIVES

In this Chapter you'll learn:

- The concept of polymorphism.
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
- How polymorphism makes systems extensible and maintainable.
- To determine an object's type at execution time.
- To declare and implement interfaces.

10.1 Introduction

- Polymorphism
  - Enables you to “program in the general” rather than “program in the specific.”
  - Polymorphism enables you to write programs that process objects that share the same superclass as if they’re all objects of the superclass; this can simplify programming.

10.1 Introduction (Cont.)

- Example: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation.
  - Each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x*-*y* coordinates. Each subclass implements method `move`.
  - A program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per second—namely, `move`.

10.1 Introduction (Cont.)

- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same message (i.e., `move`) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

10.1 Introduction (Cont.)

- With polymorphism, we can design and implement systems that are easily *extensible*
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

10.2 Polymorphism Examples

- Example: Quadrilaterals
  - If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
  - Any operation that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`.
  - These operations can also be performed on other `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.
  - Polymorphism: superclass `Quadrilateral` variable—at execution time occurs when a program invokes a method through a, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

10.3 Demonstrating Polymorphic Behavior

- In the next example, we aim a superclass reference at a subclass object.
  - Invoking a method on a subclass object via a superclass reference invokes the subclass functionality
  - The type of the referenced object, not the type of the variable, determines which method is called
- This example demonstrates that an object of a subclass can be treated as an object of its superclass, enabling various interesting manipulations.
- A program can create an array of superclass variables that refer to objects of many subclass types.
  - Allowed because each subclass object *is an* object of its superclass.

10.3 Demonstrating Polymorphic Behavior (Cont.)

- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.

```
1  // Fig. 10.1: PolymorphismTest.java
2  // Assigning superclass and subclass references to superclass and
3  // subclass variables.
4
5  public class PolymorphismTest
6  {
7     public static void main( String[] args )
8     {
9        // assign superclass reference to superclass variable
10       CommissionEmployee commissionEmployee = new CommissionEmployee(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // assign subclass reference to subclass variable
14       BasePlusCommissionEmployee basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18       // invoke toString on superclass object using superclass variable
19       System.out.printf( "%s %s:\n\n%s\n\n",
20          "Call CommissionEmployee's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString() );
22
```

Variable refers to a `CommissionEmployee` object, so that class's `toString` method is called

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 3.)

```
23         // invoke toString on subclass object using subclass variable
24         System.out.printf( "%s %s:\n\n%s\n\n",
25            "Call BasePlusCommissionEmployee's toString with subclass",
26            "reference to subclass object",
27            basePlusCommissionEmployee.toString() );
28
29         // invoke toString on subclass object using superclass variable
30         CommissionEmployee commissionEmployee2 =
31            basePlusCommissionEmployee;
32         System.out.printf( "%s %s:\n\n%s\n",
33            "Call BasePlusCommissionEmployee's toString with superclass",
34            "reference to subclass object", commissionEmployee2.toString() );
35      } // end main
36   } // end class PolymorphismTest
```

Variable refers to a `BasePlus-CommissionEmployee` object, so that class's `toString` method is called

Variable refers to a `BasePlus-CommissionEmployee` object, so that class's `toString` method is called

```
Call CommissionEmployee's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 3.)

```
Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 3 of 3.)

See code Fig10_01

10.3 Demonstrating Polymorphic Behavior (Cont.)

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.
  - The Java compiler allows this "crossover" because an object of a subclass *is an object of its superclass (but not vice versa).*
- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type.
  - If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use.
  - This process is called **dynamic binding.**

10.4 Abstract Classes and Methods

- Abstract classes
  - Sometimes it's useful to declare classes for which you never intend to create objects.
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called abstract superclasses.
  - Cannot be used to instantiate objects—abstract classes are incomplete.
  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

10.4 Abstract Classes and Methods (Cont.)

- Classes that can be used to instantiate objects are called concrete classes.
- Such classes provide implementations of every method they declare (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.

10.4 Abstract Classes and Methods (Cont.)

- You make a class abstract by declaring it with keyword abstract.
- An abstract class normally contains one or more abstract methods.
  - An abstract method is one with keyword `abstract` in its declaration, as in

```
public abstract void draw(); // abstract method
```

- Abstract methods do not provide implementations.
- A class that contains abstract methods must be an abstract class even if that class contains some concrete (nonabstract) methods.
- Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods.
- Constructors and `static` methods cannot be declared `abstract`.

10.4 Abstract Classes and Methods (Cont.)

- Cannot instantiate objects of abstract superclasses, but you can use abstract superclasses to declare variables
  - These can hold references to objects of any concrete class derived from those abstract superclasses.
  - Programs typically use such variables to manipulate subclass objects polymorphically.
- Can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

10.4 Abstract Classes and Methods (Cont.)

- Polymorphism is particularly effective for implementing so-called *layered software systems*.
- Example: Operating systems and device drivers.
  - Commands to read or write data from and to devices may have a certain uniformity.

10.5 Case Study: Payroll System Using Polymorphism

- Use an abstract method and polymorphism to perform payroll calculations based on the type of inheritance hierarchy headed by an employee.
- Enhanced employee inheritance hierarchy requirements:
  - 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 (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, **commission employees** are paid a percentage of their sales and **base-salaried commission** employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base salaried-commission employees by adding 10% to their base salaries. The company wants to write a Java application that performs its payroll calculations polymorphically.

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

- `abstract` class `Employee` represents the general concept of an employee.
- Subclasses: `SalariedEmployee`, `CommissionEmployee`, `HourlyEmployee` and `BasePlusCommissionEmployee` (an indirect subclass)
- Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application.
- Abstract class names are italicized in the UML.

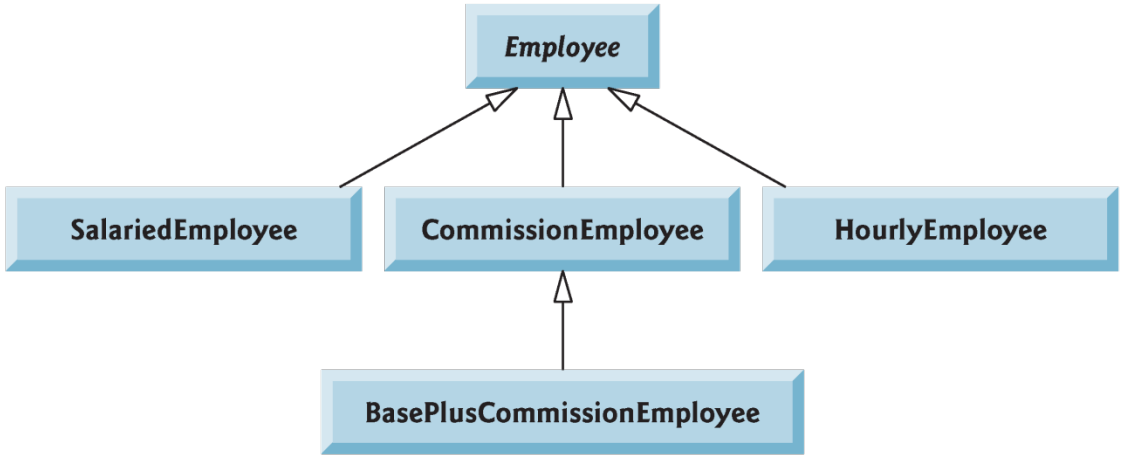

Fig. 10.2 | `Employee` hierarchy UML class diagram.

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

- Abstract superclass `Employee` declares the "interface" to the hierarchy—that is, the set of methods that a program can invoke on all `Employee` objects.
  - We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any `Employee` subclass.
- Each employee has a first name, a last name and a social security number defined in abstract superclass `Employee`.

10.5.1 Abstract Superclass `Employee`

- Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString`, in addition to the *get* and *set* methods that manipulate `Employee`'s instance variables.
- An `earnings` method applies to all employees, but each earnings calculation depends on the employee's class.
  - An `abstract` method—there is not enough information to determine what amount `earnings` should return.
  - Each subclass overrides `earnings` with an appropriate implementation.
- Iterate through the array of `Employee`s and call method `earnings` for each `Employee` subclass object.
  - Method calls processed polymorphically.

10.5.1 Abstract Superclass `Employee` (Cont.)

- The diagram in Fig. 10.3 shows each of the five classes in the hierarchy down the left side and methods `earnings` and `toString` across the top.
- For each class, the diagram shows the desired results of each method.
- Declaring the `earnings` method `abstract` indicates that each concrete subclass must provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

	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
	else 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. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

Abstract Class Employee
Concrete Subclass `SalariedEmployee`

- Fig 10_04_09 code

10.5.3 Concrete Subclass HourlyEmployee

- Fig 10_04_09 code

10.5.4 Concrete Subclass `CommissionEmployee`

- Fig 10_04_09 code

10.5.5 Indirect Concrete Subclass `BasePlusCommissionEmployee`

- Fig 10_04_09 code

```
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. 10.9 | `Employee` hierarchy test program. (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. 10.9 | `Employee` hierarchy test program. (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. 10.9 | `Employee` hierarchy test program. (Part 7 of 7.)

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting

- Fig. 10.9 creates an object of each of the four concrete.
  - Manipulates these objects nonpolymorphically, via variables of each object's own type, then polymorphically, using an array of `Employee` variables.
- While processing the objects polymorphically, the program increases the base salary of each `BasePlusCommissionEmployee` by 10%
  - Requires determining the object's type at execution time.
- Finally, the program polymorphically determines and outputs the type of each object in the `Employee` array.

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting (Cont.)

- All calls to method `toString` and `earnings` are resolved at execution time, based on the type of the object to which `currentEmployee` refers.
  - Known as dynamic binding or late binding.
  - Java decides which class's `toString` method to call at execution time rather than at compile time
- A superclass reference can be used to invoke only methods of the superclass—the subclass method implementations are invoked polymorphically.
- Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.

Software Engineering Observation 10.4

If a subclass object's reference has been assigned to a variable of one of its direct or indirect superclasses at execution time, it's acceptable to downcast the reference stored in that superclass variable back to a subclass-type reference. Before performing such a cast, use the `instanceof` *operator to ensure that the object is indeed an object of an appropriate subclass.*

Common Programming Error 10.4

When downcasting a reference, a `ClassCastException` *occurs if the referenced object at execution time does not have an* is-a *relationship with the type specified in the cast operator.*

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting (Cont.)

- Every object in Java knows its own class and can access this information through the getClass method, which all classes inherit from class `Object`.
  - The `getClass` method returns an object of type Class (from package `java.lang`), which contains information about the object's type, including its class name.
  - The result of the `getClass` call is used to invoke getName to get the object's class name.

10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables

- There are four ways to assign superclass and subclass references to variables of superclass and subclass types.
- Assigning a superclass reference to a superclass variable is straightforward.
- Assigning a subclass reference to a subclass variable is straightforward.
- Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an object of its superclass*.

10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables (Cont.)

- Attempting to assign a superclass reference to a subclass variable is a compilation error.
  - To avoid this error, the superclass reference must be cast to a subclass type explicitly.
  - At *execution time, if the object to which the reference refers is not a subclass object, an exception will occur.*
  - Use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.

10.6 `final` Methods and Classes

- A final method in a superclass cannot be overridden in a subclass.
  - Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass.
  - Methods that are declared `static` are implicitly `final`.
  - A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as static binding.

10.6 `final` Methods and Classes (Cont.)

- A final class cannot be a superclass (i.e., a class cannot extend a `final` class).
  - All methods in a `final` class are implicitly `final`.

Introduction: Interface

- An interface describes a set of methods that can be called on an object, but does not provide concrete implementations for all the methods.
- You can declare classes that implement (i.e., provide concrete implementations for the methods of) one or more interfaces.
- Each interface method must be declared in all the classes that explicitly implement the interface.

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

- Interfaces offer a capability requiring that unrelated classes implement a set of common methods.
- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
  - Example: The controls on a radio serve as an interface between radio users and a radio's internal components.
  - Can perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM)
  - Different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands).

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

- The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.
- A Java interface describes a set of methods that can be called on an object.

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

- An interface declaration begins with the keyword `interface` and contains only constants and `abstract` methods.
  - All interface members must be `public`.
  - Interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
  - All methods declared in an interface are implicitly `public abstract` methods.
  - All fields are implicitly `public`, `static` and `final`.

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

- To use an interface, a concrete class must specify that it implements the interface and must declare each method in the interface with specified signature.
  - Add the `implements` keyword and the name of the interface to the end of your class declaration's first line.
- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.
- Implementing an interface is like signing a contract with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class `abstract`."

Class Interfaces

- Now consider different classes that implement this interface
  - They will each have the same behaviors
  - Nature of the behaviors will be different
- Each of the classes implements the behaviors/methods differently

Java Interfaces

- A program component that contains headings for a number of public methods
  - Will include comments that describe the methods
- Interface can also define public named constants
- View example interface, interface Measurable

Implementing an Interface

- To implement a method, a class must
  - Include the phrase

    ```
    implements Interface_name
    ```

  - Define each specified method
- View sample class, listing 8.8

  ```
  class Rectangle implements Measurable
  ```

- View another class, listing 8.9 which also implements Measurable
  class Circle

LISTING 8.7 A Java Interface

```
/**
 An interface for methods that return
 the perimeter and area of an object.
*/
public interface Measurable
{
    /** Returns the perimeter. */
    public double getPerimeter();
    /** Returns the area. */
    public double getArea();
}
```

Do not forget the semicolons at the end of the method headings.

LISTING 8.8 An Implementation of the Interface Measurable

```
/**
A class of rectangles.
*/
public class Rectangle implements Measurable
{
    private double myWidth;
    private double myHeight;

    public Rectangle(double width, double height)
    {
        myWidth = width;
        myHeight = height;
    }
    public double getPerimeter()
    {
        return 2 * (myWidth + myHeight);
    }
    public double getArea()
    {
        return myWidth * myHeight;
    }
}
```

LISTING 8.9 Another Implementation of the Interface Measurable

```
/**
A class of circles.
*/
public class Circle implements Measurable
{
    private double myRadius;
    public Circle(double radius)
    {
        myRadius = radius;
    }
    public double getPerimeter()
    {
        return 2 * Math.PI * myRadius;
    }
    public double getCircumference()
    {
        return getPerimeter();
    }
    public double getArea()
    {
        return Math.PI * myRadius * myRadius;
    }
}
```

This method is not declared in the interface.

Calls another method instead of repeating its body

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

- An interface is often used when disparate (i.e., unrelated) classes need to share common methods and constants.
  - Allows objects of unrelated classes to be processed polymorphically by responding to the same method calls.
  - You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality.

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

- An interface is often used in place of an `abstract` class when there is no default implementation to inherit—that is, no fields and no default method implementations.

Introduction: Interface

- Once a class implements an interface, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface.
- This is true of all subclasses of that class as well.
- Interfaces are particularly useful for assigning common functionality to possibly unrelated classes.
  - Allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls.

10.7 Case Study: Creating and Using Interfaces

- Our next example reexamines the payroll system of Section 10.5.
- Suppose that the company involved wishes to perform several accounting operations in a single accounts payable application
  - Calculating the earnings that must be paid to each employee
  - Calculate the payment due on each of several invoices (i.e., bills for goods purchased)
- Both operations have to do with obtaining some kind of payment amount.
  - For an employee, the payment refers to the employee's earnings.
  - For an invoice, the payment refers to the total cost of the goods listed on the invoice.

10.7.1 Developing a `Payable` Hierarchy

- Next example builds an application that can determine payments for employees and invoices alike.
  - Classes `Invoice` and `Employee` both represent things for which the company must be able to calculate a payment amount.
  - Both classes implement the `Payable` interface, so a program can invoke method `getPaymentAmount` on `Invoice` objects and `Employee` objects alike.
  - Enables the polymorphic processing of `Invoice`s and `Employee`s.

10.7.1 Developing a Payable Hierarchy (Cont.)

- Fig. 10.10 shows the accounts payable hierarchy.
- The UML distinguishes an interface from other classes by placing «interface» above the interface name.
- The UML expresses the relationship between a class and an interface through a realization.
  - A class is said to "realize," or implement, the methods of an interface.
  - A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface.

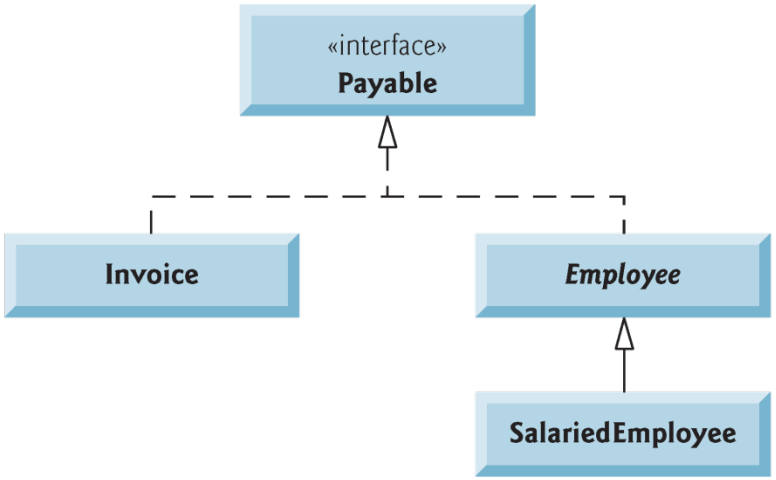

Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

```
1  // Fig. 10.11: Payable.java
2  // Payable interface declaration.
3
4  public interface Payable
5  {
6     double getPaymentAmount(); // calculate payment; no implementation
7  } // end interface Payable
```

Fig. 10.11 | `Payable` interface declaration.

10.7.4 Modifying Class `Employee` to Implement Interface `Payable`

- When a class implements an interface, it makes a contract with the compiler
  - The class will implement each of the methods in the interface or that the class will be declared `abstract`.
  - If the latter, we do not need to declare the interface methods as `abstract` in the `abstract` class—they are already implicitly declared as such in the interface.
  - Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the contract.
  - If the subclass does not do so, it too must be declared `abstract`.
- Each direct `Employee` subclass inherits the superclass's contract to implement method `getPaymentAmount` and thus must implement this method to become a concrete class for which objects can be instantiated.

10.7.5 Modifying Class SalariedEmployee for Use in the Payable Hierarchy

- A modified `SalariedEmployee` class that extends `Employee` and fulfills superclass `Employee`'s contract to implement `Payable` method `getPayment-Amount`.

Class Invoice & Employee & SalariedEmployee

- See code
- Fig 10_11_15

10.7.5 Modifying Class SalariedEmployee for Use in the Payable Hierarchy (Cont.)

- Objects of any subclasses of a class that `implements` an interface can also be thought of as objects of the interface type.
- Thus, just as we can assign the reference of a `SalariedEmployee` object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object to an interface `Payable` variable.
- `Invoice` implements `Payable`, so an `Invoice` object also *is a* `Payable` *object, and we can assign the reference of an* `Invoice` *object to a* `Payable` *variable.*

10.7.6 Using Interface Payable to Process Invoices and Employees Polymorphically

- See Code
- Fig 10_11_15

```
1  // Fig. 10.15: PayableInterfaceTest.java
2  // Tests interface Payable.
3
4  public class PayableInterfaceTest
5  {
6     public static void main( String[] args )
7     {
8        // create four-element Payable array
9        Payable[] payableObjects = new Payable[ 4 ];
10
11       // populate array with objects that implement Payable
12       payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13       payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14       payableObjects[ 2 ] =
15          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16       payableObjects[ 3 ] =
17          new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19       System.out.println(
20          "Invoices and Employees processed polymorphically:\n" );
21
```

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 1 of 3.)

```
22         // generically process each element in array payableObjects
23         for ( Payable currentPayable : payableObjects )
24         {
25            // output currentPayable and its appropriate payment amount
26            System.out.printf( "%s \n%s: $%,.2f\n\n",
27               currentPayable.toString(),
28               "payment due", currentPayable.getPaymentAmount() );
29         } // end for
30      } // end main
31   } // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

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

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 2 of 3.)

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

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

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

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 3 of 3.)

10.7.3

- Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs.
- To implement more than one interface, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class ClassName extends SuperclassName
   implements FirstInterface, SecondInterface, …
```

10.7.7 Common Interfaces of the Java API

- The Java API's (Application Programming Interfaces) interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program.
- A few of the more popular interfaces of the Java API(Comparable, GUI event listener interface….).