Chapter 10 Object-Oriented Programming: Polymorphism and Interfaces

Java™ How to Program, 10/e

10.9 Creating and Using Interfaces

- Our next example reexamines a payroll system
- 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.

10.9 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).

10.9 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.

10.9 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`.

Good Programming Practice 10.1

According to the Java Language Specification, *it's proper style to declare an interface's* `abstract` *methods without keywords* `public` *and* `abstract`, *because they're redundant in interface-method declarations. Similarly, an interface's constants should be declared without keywords* `public`, `static` *and* `final`, *because they, too, are redundant.*

10.9 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`."

Common Programming Error 10.6

Failing to implement any method of an interface in a concrete class that `implements` *the interface results in a compilation error indicating that the class must be declared* `abstract`.

10.9 Creating and Using Interfaces (Cont.)

- An interface is often used when disparate classes (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.

10.9 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.
- A `public` interface must be declared in a file with the same name as the interface and the `.java` filename extension.

Software Engineering Observation 10.7

Many developers feel that interfaces are an even more important modeling technology than classes, especially with the new interface enhancements in Java SE 8 (see Section 10.10).

10.9.1 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.

10.9.1 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 subclass inherits its superclass's realization relationships.

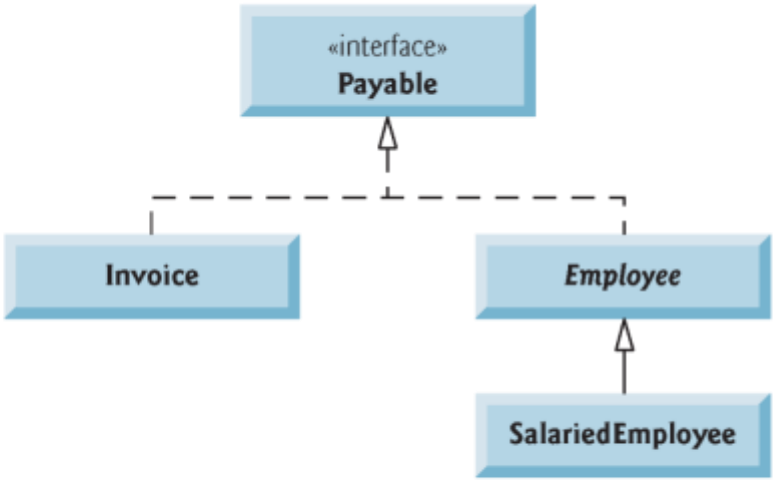

Fig. 10.10 | `Payable` hierarchy UML class diagram.

10.9.2 Interface Payable

- Fig. 10.11 shows the declaration of interface `Payable`.
- Interface methods are always `public` and `abstract`, so they do not need to be declared as such.
- Interfaces can have any number of methods.
- Interfaces may also contain `final` and `static` constants

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

Fig. 10.11 | Payable interface declaration.

10.9.3 Class `Invoice`

- 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, ...
```

```
1  // Fig. 10.12: Invoice.java
2  // Invoice class that implements Payable.
3
4  public class Invoice implements Payable
5  {
6     private final String partNumber;
7     private final String partDescription;
8     private int quantity;
9     private double pricePerItem;
10
11    // constructor
12    public Invoice(String partNumber, String partDescription, int quantity,
13       double pricePerItem)
14    {
15       if (quantity < 0) // validate quantity
16          throw new IllegalArgumentException("Quantity must be >= 0");
17
18       if (pricePerItem < 0.0) // validate pricePerItem
19          throw new IllegalArgumentException(
20             "Price per item must be >= 0");
21
```

Fig. 10.12 | Invoice class that implements Payable. (Part 1 of 4.)

```
22          this.quantity = quantity;
23          this.partNumber = partNumber;
24          this.partDescription = partDescription;
25          this.pricePerItem = pricePerItem;
26       } // end constructor
27
28       // get part number
29       public String getPartNumber()
30       {
31          return partNumber; // should validate
32       }
33
34       // get description
35       public String getPartDescription()
36       {
37          return partDescription;
38       }
39
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 2 of 4.)

```
40      // set quantity
41      public void setQuantity(int quantity)
42      {
43         if (quantity < 0) // validate quantity
44            throw new IllegalArgumentException("Quantity must be >= 0");
45
46         this.quantity = quantity;
47      }
48
49      // get quantity
50      public int getQuantity()
51      {
52         return quantity;
53      }
54
55      // set price per item
56      public void setPricePerItem(double pricePerItem)
57      {
58         if (pricePerItem < 0.0) // validate pricePerItem
59            throw new IllegalArgumentException(
60               "Price per item must be >= 0");
61
62         this.pricePerItem = pricePerItem;
63      }
```

Fig. 10.12 | Invoice class that implements Payable. (Part 3 of 4.)

```
64
65      // get price per item
66      public double getPricePerItem()
67      {
68         return pricePerItem;
69      }
70
71      // return String representation of Invoice object
72      @Override
73      public String toString()
74      {
75         return String.format("%s: %n%s: %s (%s) %n%s: %d %n%s: $%,.2f",
76            "invoice", "part number", getPartNumber(), getPartDescription(),
77            "quantity", getQuantity(), "price per item", getPricePerItem());
78      }
79
80      // method required to carry out contract with interface Payable
81      @Override
82      public double getPaymentAmount()
83      {
84         return getQuantity() * getPricePerItem(); // calculate total cost
85      }
86   } // end class Invoice
```

Fig. 10.12 | Invoice class that implements Payable. (Part 4 of 4.)

Software Engineering Observation 10.8

All objects of a class that implements multiple interfaces have the is-a *relationship with each implemented interface type.*

10.9.4 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* method in the interface or the class will be declared `abstract`.
  - Because class `Employee` does not provide a `getPaymentAmount` method, the class must be declared `abstract`.
  - 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.

```
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass that implements Payable.
3
4  public abstract class Employee implements Payable
5  {
6     private final String firstName;
7     private final String lastName;
8     private final String socialSecurityNumber;
9
10    // constructor
11    public Employee(String firstName, String lastName,
12       String socialSecurityNumber)
13    {
14       this.firstName = firstName;
15       this.lastName = lastName;
16       this.socialSecurityNumber = socialSecurityNumber;
17    }
18
19    // return first name
20    public String getFirstName()
21    {
22       return firstName;
23    }
```

Fig. 10.13 | Employee abstract superclass that implements Payable. (Part 1 of 2.)

```
24
25      // return last name
26      public String getLastName()
27      {
28         return lastName;
29      }
30
31      // return social security number
32      public String getSocialSecurityNumber()
33      {
34         return socialSecurityNumber;
35      }
36
37      // return String representation of Employee object
38      @Override
39      public String toString()
40      {
41         return String.format("%s %s%nsocial security number: %s",
42            getFirstName(), getLastName(), getSocialSecurityNumber());
43      }
44
45      // Note: We do not implement Payable method getPaymentAmount here so
46      // this class must be declared abstract to avoid a compilation error.
47   } // end abstract class Employee
```

Fig. 10.13 | `Employee` abstract superclass that implements `Payable`. (Part 2 of 2.)

10.9.5 Modifying Class SalariedEmployee for Use in the Payable Hierarchy

- Figure 10.14 contains a modified `SalariedEmployee` class that extends `Employee` and fulfills superclass `Employee`'s contract to implement `Payable` method `getPayment-Amount`.

```
1  // Fig. 10.14: SalariedEmployee.java
2  // SalariedEmployee class that implements interface Payable.
3  // method getPaymentAmount.
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // constructor
9     public SalariedEmployee(String firstName, String lastName,
10       String socialSecurityNumber, double weeklySalary)
11    {
12       super(firstName, lastName, socialSecurityNumber);
13
14       if (weeklySalary < 0.0)
15          throw new IllegalArgumentException(
16             "Weekly salary must be >= 0.0");
17
18       this.weeklySalary = weeklySalary;
19    }
20
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 1 of 3.)

```
21      // set salary
22      public void setWeeklySalary(double weeklySalary)
23      {
24         if (weeklySalary < 0.0)
25            throw new IllegalArgumentException(
26               "Weekly salary must be >= 0.0");
27
28         this.weeklySalary = weeklySalary;
29      }
30
31      // return salary
32      public double getWeeklySalary()
33      {
34         return weeklySalary;
35      }
36
37      // calculate earnings; implement interface Payable method that was
38      // abstract in superclass Employee
39      @Override
40      public double getPaymentAmount()
41      {
42         return getWeeklySalary();
43      }
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 2 of 3.)

```
44
45      // return String representation of SalariedEmployee object
46      @Override
47      public String toString()
48      {
49         return String.format("salaried employee: %s%n%s: $%,.2f",
50            super.toString(), "weekly salary", getWeeklySalary());
51      }
52   } // end class SalariedEmployee
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 3 of 3.)

Software Engineering Observation 10.9

When a method parameter is declared with a superclass or interface type, the method processes the object passed as an argument polymorphically.

Software Engineering Observation 10.10

Using a superclass reference, we can polymorphically invoke any method declared in the superclass and its superclasses (e.g., class `Object`*). Using an interface reference, we can polymorphically invoke any method declared in the interface, its superinterfaces (one interface can extend another) and in class* `Object`*—a variable of an interface type must refer to an object to call methods, and all objects have the methods of class* `Object`*.*

10.9.5 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.

10.9.6 Using Interface Payable to Process Invoices and Employees Polymorphically

- `PayableInterfaceTest` (Fig. 10.15) illustrates that interface `Payable` can be used to process a set of `Invoice`s and `Employee`s *polymorphically* in a single application.

```
1  // Fig. 10.15: PayableInterfaceTest.java
2  // Payable interface test program processing Invoices and
3  // Employees polymorphically.
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:");
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("%n%s %n%s: $%,.2f%n",
27               currentPayable.toString(), // could invoke implicitly
28               "payment due", currentPayable.getPaymentAmount());
29         }
30      } // end main
31   } // end class PayableInterfaceTest
```

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

API Example

- To use sort method of Arrays class, implement Comparable interface

Tagging Interfaces

- A *tagging interface* (also called a *marker interface*) is an empty interface that have *no* methods or constant values. They're used to add *is-a* relationships to classes.
- For example, Java has a mechanism called *object serialization*, which can convert objects to byte representations and can convert those byte representations back to objects, using classes ObjectOutputStream and ObjectInputStream.

- To enable this mechanism to work with your objects, you simply have to *tag* them as Serializable by adding implements Serializable to the end of your class declaration's first line. Then all the objects of your class have the *is-a* relationship with Serializable—that's all it takes to implement basic object serialization.

10.9.7 Some Common Interfaces of the Java API

- You'll use interfaces extensively when developing Java applications. The Java API contains numerous interfaces, and many of the Java API methods take interface arguments and return interface values.
- Figure 10.16 overviews a few of the more popular interfaces of the Java API that we use in later chapters.

Interface	Description
`Comparable`	Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators *cannot* be used to compare objects. Interface `Comparable` is used to allow objects of a class that `implements` the interface to be compared to one another. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 16, Generic Collections, and Chapter 20, Generic Classes and Methods.
`Serializable`	An interface used to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 15, Files, Streams and Object Serialization, and Chapter 28, Networking.
`Runnable`	Implemented by any class that represents a task to perform. Objects of such as class are often executed in parallel using a technique called *multithreading* (discussed in Chapter 23, Concurrency). The interface contains one method, `run`, which specifies the behavior of an object when executed.

Fig. 10.16 | Common interfaces of the Java API. (Part 1 of 2.)

Interface	Description
GUI event-listener interfaces	You work with graphical user interfaces (GUIs) every day. In your web browser, you might type the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. Your interaction is known as an *event*, and the code that the browser uses to respond to an event is known as an *event handler*. In Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2, you'll learn how to build GUIs and event handlers that respond to user interactions. Event handlers are declared in classes that implement an appropriate *event-listener interface*. Each event-listener interface specifies one or more methods that must be implemented to respond to user interactions.
`AutoCloseable`	Implemented by classes that can be used with the `try`-with-resources statement (Chapter 11, Exception Handling: A Deeper Look) to help prevent resource leaks.

Fig. 10.16 | Common interfaces of the Java API. (Part 2 of 2.)

10.10 Java SE 8 Interface Enhancements

- This section introduces Java SE 8's new interface features.
- We discuss these in more detail in later chapters.

10.10.1 `default` Interface Methods

- Prior to Java SE 8, interface methods could be *only* `public abstract` methods.
  - An interface specified *what* operations an implementing class must perform but not *how* the class should perform them.
- In Java SE 8, interfaces also may contain `public default` methods with concrete default implementations that specify how operations are performed when an implementing class does not override the methods.
- If a class implements such an interface, the class also receives the interface's `default` implementations (if any).
- To declare a default method, place the keyword `default` before the method's return type and provide a concrete method implementation.

10.10.1 `default` Interface Methods (Cont.)

Adding Methods to Existing Interfaces

- Any class that implements the original interface will *not* break when a `default` method is added.
  - The class simply receives the new default method.
- When a class implements a Java SE 8 interface, the class "signs a contract" with the compiler that says,
  - "I will declare all the *`abstract`* methods specified by the interface or I will declare my class `abstract`"
- The implementing class is not required to override the interface's `default` methods, but it can if necessary.

10.10.1 `default` Interface Methods (Cont.)

Interfaces vs. `abstract` Classes

- Prior to Java SE 8, an interface was typically used (rather than an `abstract` class) when there were no implementation details to inherit—no fields and no method implementations.
- With `default` methods, you can instead declare common method implementations in interfaces, which gives you more flexibility in designing your classes.

10.10.2 `static` Interface Methods (Cont.)

- Prior to Java SE 8, it was common to associate with an interface a class containing `static` helper methods for working with objects that implemented the interface.
- In Chapter 16, you'll learn about class `Collections` which contains many `static` helper methods for working with objects that implement interfaces `Collection`, `List`, `Set` and more.
- `Collections` method `sort` can sort objects of *any* class that implements interface `List`.
- With `static` interface methods, such helper methods can now be declared directly in interfaces rather than in separate classes.

10.10.3 Functional Interfaces

- As of Java SE 8, any interface containing only one `abstract` method is known as a functional interface.
- Functional interfaces that you'll use in this book include:
  - `ActionListener` (Chapter 12)—You'll implement this interface to define a method that's called when the user clicks a button.
  - `Comparator` (Chapter 16)—You'll implement this interface to define a method that can compare two objects of a given type to determine whether the first object is less than, equal to or greater than the second.
  - `Runnable` (Chapter 23)—You'll implement this interface to define a task that may be run in parallel with other parts of your program.

Java SE 9 private Interface Methods

- As you know, a class's private helper methods may be called only by the class's other methods.
- As of Java SE 9, you can declare helper methods in *interfaces* via **private interface methods**.
- An interface's private instance methods can be called directly (i.e., without an object reference) only by the interface's other instance methods. An interface's private static methods can be called by any of the interface's instance or static methods.

Common Programming Error 10.7

Including the `default` *keyword in a* `private` *interface method's declaration is a compilation error—*`default` *methods must be* `public`.

Java SE 9 private Interface Methods

- In Java 9 and later versions, an interface can have six different things:

1. Constant variables
2. Abstract methods
3. Default methods
4. Static methods
5. Private methods
6. Private Static methodsThese private methods will improve code re-usability inside interfaces and will provide choice to expose only our intended methods implementations to users. These methods are only accessible within that interface only and cannot be accessed or inherited from an interface to another interface or class.

private Constructors

- ***Preventing Object Instantiation***
  - You can prevent client code from creating objects of a class by making the class's constructors private. For example, consider class Math, which contains only public static constants and public static methods. There's no need to create a Math object to use the class's constants and methods, so its constructor is private.
- ***Sharing Initialization Code in Constructors***
  - One common use of a private constructor is sharing initialization code among class's other constructors. You can use delegating constructors to call the private constructor that contains the shared initialization code.
- ***Factory Methods***
  - ***Discussed later***

Interface inheritance vs Implementation inheritance

- Extends - Implementation inheritance is a relationship where a child class inherits behaviour implementation from a base class.
- Implements - Interface inheritance is when a child class only inherits the description of behaviour from the base class and provides the implementation itself.