10.9 Creating and Using Interfaces

8

[Note: Java SE 8 interface enhancements are introduced in Section 10.10 and discussed in more detail in Chapter 17. Java SE 9 interface enhancements are introduced in Section 10.11.] Our next example (Figs. 10.11–10.14) 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—in addition to calculating the earnings that must be paid to each employee, the company must also calculate the payment due on each of several invoices (i.e., bills for goods purchased). Though applied to *unrelated* things—employees and invoices—both operations have to do with obtaining a payment amount. For an employee, the payment is the employee’s earnings. For an invoice, the payment is the total cost of the goods listed on the invoice. Can we calculate such *different* things as the payments due for employees and invoices in *a single* application *polymorphically*? Does Java offer a capability requiring that *unrelated* classes implement a set of *common* methods (e.g., a method that calculates a payment amount)? Java **interfaces** offer exactly this capability.

Standardizing Interactions

Interfaces define and standardize the ways in which things such as people and systems can interact with one another. For example, the controls on a radio serve as an interface between radio users and a radio's internal components. The controls allow users to perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM), and different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands). The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.

Similarly, in our car analogy from Section 1.5, a "basic-driving-capabilities" interface consisting of a steering wheel, an accelerator pedal and a brake pedal would enable a driver to tell the car *what* to do. Once you know how to use this interface for turning, accelerating and braking, you can drive many types of cars, even though manufacturers may *implement* these systems *differently*. For example, there are many types of braking systems—disc brakes, drum brakes, antilock brakes, hydraulic brakes, air brakes and more. When you press the brake pedal, your car's actual brake system is irrelevant—all that matters is that the car slows down when you press the brake.

Software Objects

Communicate Via Interfaces

Software objects also communicate via interfaces. A Java interface describes a set of methods that can be called on an object to tell it, for example, to perform some task or return some piece of information. The next example introduces an interface named `Payable` to describe the functionality of any object that must be "capable of being paid" and thus must offer a method to determine the proper payment amount due. An **interface declaration** begins with the keyword `interface` and contains *only* constants and `abstract` methods. Unlike classes, all interface members *must* be `public`, and *interfaces may not specify any implementation details*, such as concrete method declarations and instance variables.[1] All methods declared in an interface are implicitly `public abstract` methods, and all fields are implicitly `public`, `static` and `final`.

[1]. Sections 10.10–10.11 introduce the Java SE 8 and Java SE 9 interface enhancements that allow method implementations and `private` methods, respectively, in interfaces.

Using an Interface

To use an interface, a concrete class must specify that it `implements` the interface and must declare each method in the interface with the signature specified in the interface declaration. To specify that a class implements an interface, add the `implements` keyword and the name of the interface to the end of your class declaration's first line, as in:

```
public class ClassName extends SuperclassName impleme
```

or

```
public class ClassName implements InterfaceName
```

InterfaceName in the preceding snippets may be a comma-separated list of interface names.

Implementing an interface is like signing a *contract* with the compiler that states, “I will declare all the `abstract` methods specified by the interface or I will declare my class `abstract`.”

Common Programming Error 10.6

In a concrete class that `implements` *an interface, failing to implement any of the interface’s* `abstract` *methods results in a compilation error indicating that the class must be declared* `abstract`.

Relating Disparate Types

An interface is often used when *disparate* classes—i.e., classes that are not related by a class hierarchy—need to share common methods and constants. This allows objects of *unrelated* classes to be processed *polymorphically*—objects of classes that implement the *same* interface can respond to the *same* method calls (for methods of that interface). You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality. For example, in the accounts payable application developed in this section, we implement interface `Payable` in any class that must be able to calculate a payment amount (e.g., `Employee` and `Invoice`).

Interfaces vs. Abstract Classes

An interface should be used in place of an `abstract` *class when there's no default implementation to inherit*—that is, no fields and no concrete method implementations. Like `public abstract` classes, interfaces are typically `public` types. Like a `public` class, a `public` interface must be declared in a file with the same name as the interface and the `.java` filename extension.

Software Engineering Observation 10.8

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

Tagging Interfaces

A *tagging interface* (also called a *marker interface*) is an empty interfaces 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.

10.9.1 Developing a `Payable` Hierarchy

To build an application that can determine payments for employees and invoices alike, we first create interface `Payable`, which contains method `getPaymentAmount` that returns a `double` amount that must be paid for an object of any class that implements the interface. Method `getPaymentAmount` is a general-purpose version of method `earnings` of the `Employee` hierarchy—method `earnings` calculates a payment amount specifically for an `Employee`, while `getPaymentAmount` can be applied to a broad range of possibly unrelated objects. After declaring interface `Payable`, we introduce class `Invoice`, which `implements` interface `Payable`. We then modify class `Employee` such that it also implements interface `Payable`.

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. As we’ll soon see, this enables the *polymorphic* processing of `Invoice`s and `Employee`s required for the company’s accounts payable application.

Good Programming Practice 10.1

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.*

UML Diagram Containing an Interface

The UML class diagram in Fig. 10.10 shows the interface and class hierarchy used in our accounts payable application. The hierarchy begins with interface `Payable`. The UML distinguishes an interface from classes by placing the word "interface" in guillemets (« and ») above the interface name. The UML expresses the relationship between a class and an interface through a relationship known as **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. The diagram in Fig. 10.10 indicates that classes `Invoice` and `Employee` each realize interface `Payable`. As in the class diagram of Fig. 10.2, class `Employee` appears in *italics*, indicating that it's an *abstract class*. *Concrete* class `SalariedEmployee` extends `Employee`, *inheriting its superclass's realization relationship* with interface `Payable`.

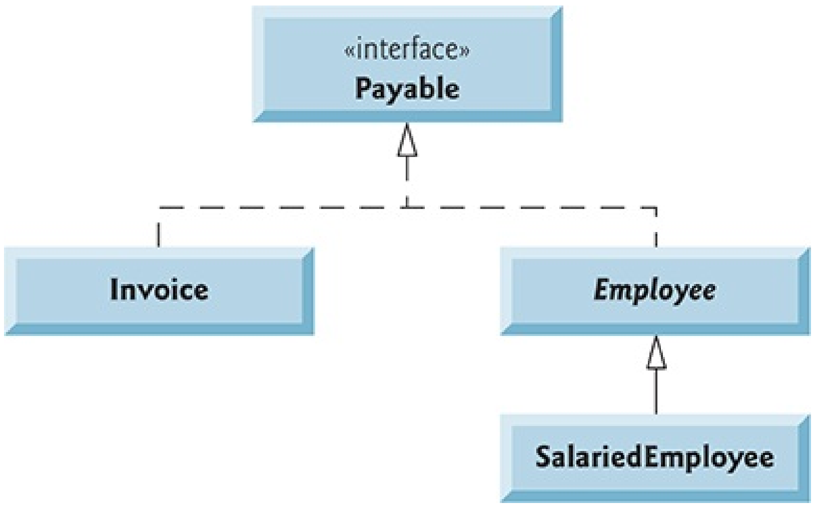

Fig. 10.10

`Payable` hierarchy UML class diagram.

10.9.2 Interface `Payable`

Interface `Payable`’s declaration begins in Fig. 10.11 at line 4 with the `interface` keyword. The interface contains the `public abstract` method named `getPaymentAmount`. Interface methods are `public` and `abstract` by default, so they do not need to be declared as such. Interface `Payable` has only one method, but interfaces can have *any* number of methods. In addition, method `getPaymentAmount` has no parameters, but interface methods *can* have parameters. Interfaces may also contain `final static` constants.

Good Programming Practice 10.2

Use `public` *and* `abstract` *explicitly when declaring interface methods to make your intentions clear. As you’ll see in Sections 10.10–10.11, Java SE 8 and Java SE 9 allow other kinds of methods in interfaces.*

```java
// Fig. 10.11: Payable.java
// Payable interface declaration.

public interface Payable {
   public abstract double getPaymentAmount(); //
}
```

Fig. 10.11

`Payable` interface declaration.

10.9.3 Class `Invoice`

Class `Invoice` (Fig. 10.12) represents a simple invoice that contains billing information for only one kind of part. The class declares `private` instance variables `partNumber`, `partDescription`, `quantity` and `pricePerItem`(in lines 5–8) that indicate the part number, a description of the part, the quantity of the part ordered and the price per item. Class `Invoice` also contains a constructor (lines 11–26), *get* methods (lines 29–38) and a `toString` method (lines 41–46) that returns a `String` representation of an `Invoice` object.

```java
1  // Fig. 10.12: Invoice.java
2  // Invoice class that implements Payable.
3
4  public class Invoice implements Payable {
5     private final String partNumber;
```

```java
    private final String partDescription;
    private final int quantity;
    private final double pricePerItem;

    // constructor
    public Invoice(String partNumber, String partD
        double pricePerItem) {
        if (quantity < 0) { // validate quantity
            throw new IllegalArgumentException("Quan
        }

        if (pricePerItem < 0.0) { // validate price
            throw new IllegalArgumentException(
                "Price per item must be >= 0");
        }

        this.quantity = quantity;
        this.partNumber = partNumber;
        this.partDescription = partDescription;
        this.pricePerItem = pricePerItem;
    }

    // get part number
    public String getPartNumber() {return partNumb

    // get description
    public String getPartDescription() {return par

    // get quantity
    public int getQuantity() {return quantity;}

    // get price per item
    public double getPricePerItem() {return priceP

    // return String representation of Invoice obj
    @Override
    public String toString() {
        return String.format("%s: %n%s: %s (%s) %n%
            "invoice", "part number", getPartNumber(
            "quantity", getQuantity(), "price per it
```

```
46     }
47
48     // method required to carry out contract with
49     @Override
50     public double getPaymentAmount() {
51        return getQuantity() * getPricePerItem(); /
52     }
53  }
```

Fig. 10.12

`Invoice` class that implements `Payable`.

Line 4 indicates that class `Invoice` implements interface `Payable`. Like all classes, class `Invoice` also *implicitly* extends `Object`. Class `Invoice` implements the one `abstract` method in interface `Payable`—method `getPaymentAmount` is declared in lines 49–52. The method calculates the total payment required to pay the invoice. The method multiplies the values of `quantity` and `pricePerItem` (obtained through the appropriate *get* methods) and returns the result. This method satisfies the implementation requirement for this method in interface `Payable`—we've *fulfilled* the *interface contract* with the compiler.

A Class Can Extend Only

One Other Class But Can Implement Many Interfaces

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 impleme
    SecondInterface, …
```

Software Engineering Observation 10.9

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

Class `ArrayList` (Section 7.16) is one of many Java API classes that implement multiple interfaces. For example, `ArrayList` implements interface `Iterable`, which enables the enhanced `for` statement to iterate over an `ArrayList`'s elements. `ArrayList` also implements interface `List` (Section 16.6), which declares the common methods (such as `add`, `remove` and `contains`) that you can call on any object that represents a lists of items.

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

We now modify class `Employee` to implement interface `Payable` (Fig. 10.13). This class declaration is identical to that of Fig. 10.4 with two exceptions:

- Line 4 of Fig. 10.13 indicates that class `Employee` now implements `Payable`.
- Line 38 implements interface `Payable`’s `getPaymentAmount` method.

Notice that `getPaymentAmount` simply calls `Employee`’s `abstract` method `earnings`. At execution time, when `getPaymentAmount` is called on an object of an `Employee` subclass, `getPaymentAmount` calls that subclass’s concrete `earnings` method, which knows how to calculate earnings for objects of that subclass type.

```
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass that implements P
3
4  public abstract class Employee implements Payable
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
```

```java

   // constructor
   public Employee(String firstName, String lastN
      String socialSecurityNumber) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.socialSecurityNumber = socialSecurityN
   }

   // return first name
   public String getFirstName() {return firstName

   // return last name
   public String getLastName() {return lastName;}

   // return social security number
   public String getSocialSecurityNumber() {retur

   // return String representation of Employee ob
   @Override
   public String toString() {
      return String.format("%s %s%nsocial securit
         getFirstName(), getLastName(), getSocial
   }

   // abstract method must be overridden by concr
   public abstract double earnings(); // no imple

   // implementing getPaymentAmount here enables
   // class hierarchy to be used in an app that p
   public double getPaymentAmount() {return earni
}
```

Fig. 10.13

`Employee` `abstract` superclass that implements

`Payable`.

Subclasses of `Employee` and Interface `Payable`

When a class implements an interface, the same *is-a* relationship as inheritance applies. Class `Employee` implements `Payable`, so we can say that an `Employee` *is a* `Payable`, and thus any object of an `Employee` subclass also *is a* `Payable`. So, if we update the class hierarchy in Section 10.5 with the new `Employee` superclass in Fig. 10.13, then `Salaried-Employee`s, `HourlyEmployee`s, `CommissionEmployee`s and `BasePlusCommissionEmployee`s are all `Payable` objects. Just as we can assign the reference of a `SalariedEmployee` subclass object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object (or any other `Employee` derived-class object) to a `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.

Software Engineering Observation 10.10

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

Software Engineering Observation 10.11

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

Software Engineering Observation 10.12

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 Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically

`PayableInterfaceTest` (Fig. 10.14) illustrates that interface `Payable` can be used to process a set of `Invoice`s and `Employee`s *polymorphically* in a single application. Lines 7–12 declare and initialize the four-element array `payableObjects`. Lines 8–9 place the references of `Invoice` objects in `payableObjects`' first two elements. Lines 10–11 then place the references of `SalariedEmployee` objects in `payableObjects`' last two elements. The elements are allowed to be initialized with `Invoice`s and `SalariedEmployee`s, because an `Invoice` *is a* `Payable`, a `SalariedEmployee` *is an* `Employee` and an `Employee` *is a* `Payable`.

```
1  // Fig. 10.14: PayableInterfaceTest.java
2  // Payable interface test program processing Invo
3  // Employees polymorphically.
4  public class PayableInterfaceTest {
5     public static void main(String[] args) {
        // create four-element Payable array
        Payable[] payableObjects = new Payable[] {
           new Invoice("01234", "seat", 2, 375.00),
           new Invoice("56789", "tire", 4, 79.95),
           new SalariedEmployee("John", "Smith", "1
           new SalariedEmployee("Lisa", "Barnes", "
        };

        System.out.println(
           "Invoices and Employees processed polymo

        // generically process each element in arra
        for (Payable currentPayable : payableObject
            // output currentPayable and its approp
            System.out.printf("%n%s %npayment due:
               currentPayable.toString(), // could
               currentPayable.getPaymentAmount());
        }
     }
  }
```

```
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. 10.14

`Payable` interface test program processing `Invoice`s and `Employee`s polymorphically.

Lines 18–23 *polymorphically* process each `Payable` object in `payableObjects`, displaying each object’s `String` representation and payment amount. Line 21 invokes method `toString` via a `Payable` interface reference, even though `toString` is not declared in interface `Payable`—*all references (including those of interface types) refer to objects that extend* `Object` *and therefore have a* `toString` *method*. (Method `toString` also can be invoked *implicitly* here.) Line 22 invokes `Payable` method `getPaymentAmount` to obtain the payment amount for each object in `payableObjects`, regardless of the actual type of the object. The output reveals that each of the method calls in lines 21–22 invokes the appropriate class’s `toString` and `getPaymentAmount` methods.

10.9.6 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 10.15 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 `ArrayList`. We use `Comparable` in Chapter 16, Generic Collections, and Chapter 20, Generic Classes and Methods: A Deeper Look.
`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.
`Runnable`	Implemented by any class that represents a task to perform. Objects of such a 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.
	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

GUI event-listener interfaces	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, JavaFX Graphical User Interfaces: Part 1, you'll begin learning how to build GUIs with 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. We use this interface in Chapter 15, Files, Input/Output Streams, NIO and XML Serialization, and Chapter 24, Accessing Databases with JDBC.

Fig. 10.15

Common interfaces of the Java API.