Chapter 10
Object-Oriented Programming: Polymorphism and Interfaces

Java How to Program, 11/e, Global Edition

Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Learn the concept of polymorphism and how it enables "programming in the general."
- Use overridden methods to effect polymorphism.
- Distinguish between abstract and concrete classes.

OBJECTIVES

- Declare abstract methods to create abstract classes.
- Learn how polymorphism makes systems extensible and maintainable.
- Determine an object's type at execution time.
- Declare and implement interfaces, and become familiar with the Java SE 8 interface enhancements.

OUTLINE

OUTLINE (cont.)

10.5 Case Study: Payroll System Using Polymorphism

- 10.5.1 Abstract Superclass `Employee`
- 10.5.2 Concrete Subclass `SalariedEmployee`
- 10.5.3 Concrete Subclass `HourlyEmployee`
- 10.5.4 Concrete Subclass `CommissionEmployee`
- 10.5.5 Indirect Concrete Subclass `BasePlusCommissionEmployee`
- 10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting

OUTLINE (cont.)

OUTLINE (cont.)

10.9 Creating and Using Interfaces

- 10.9.1 Developing a `Payable` Hierarchy
- 10.9.2 Interface `Payable`
- 10.9.3 Class `Invoice`
- 10.9.4 Modifying Class `Employee` to Implement Interface `Payable`
- 10.9.5 Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically
- 10.9.6 Some Common Interfaces of the Java API

OUTLINE (cont.)

OUTLINE (cont.)

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 were 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 new classes simply "plug right in."
  - 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.

10.1 Introduction (Cont.)

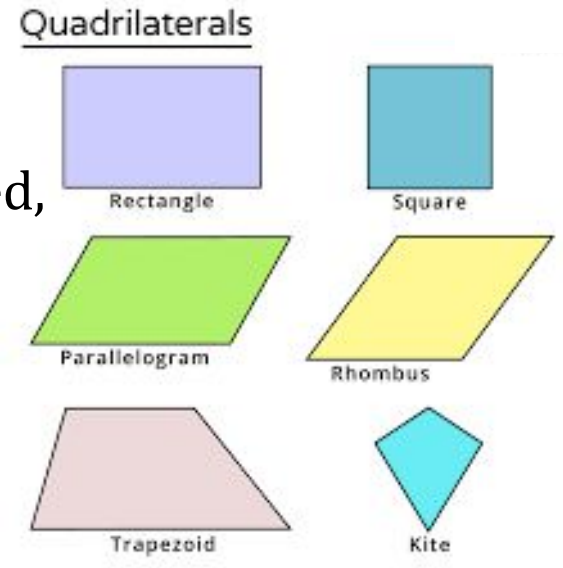
- Once a class implements an interface, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface.
- This is true of all subclasses of that class as well.
- Interfaces are particularly 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.

10.1 Introduction (Cont.)

- The chapter continues with an introduction to Java *interfaces*, which are particularly useful for assigning *common* functionality to possibly unrelated classes.
- This allows objects of these classes to be processed polymorphically—objects of classes that implement the *same* interface can respond to all of the interface method calls.

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 `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`.
  - Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—
    at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.



10.2 Polymorphism Examples (Cont.)

- Example: Space Objects in a Video Game
  - A video game manipulates objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`.
    - Each inherits from `SpaceObject` and overrides its `draw` method.
  - A screen manager maintains a collection of references to objects of the various classes and periodically sends each object the same message—namely, `draw`.
  - Each object responds in a unique way.
    - A `Martian` object might draw itself in red with green eyes and the appropriate number of antennae.
    - A `SpaceShip` object might draw itself as a bright silver flying saucer.
    - A `LaserBeam` object might draw itself as a bright red beam across the screen.
  - The same message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

10.2 Polymorphism Examples (Cont.)

- A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code.
- To add new objects to our video game:
  - Build a class that extends `SpaceObject` and provides its own `draw` method implementation.
  - When objects of that class appear in the `SpaceObject` collection, the screen-manager code *invokes method `draw`, exactly as it does for every other object in the collection, regardless of its type.*
  - So the new objects simply "**plug right in**" **without** any **modification** of the screen manager code by the programmer.

Software Engineering Observation 10.1

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics. You can tell objects to behave in manners appropriate to those objects, without knowing their specific types, as long as they belong to the same inheritance hierarchy.

Software Engineering Observation 10.2

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

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.

Software Engineering Observation 10.3

Although it's allowed, you should generally avoid downcasting.

```
1  // Fig. 10.1: PolymorphismTest.java
2  // Assigning superclass and subclass references to superclass and
3  // subclass variables.
4
5  public class PolymorphismTest {
6     public static void main(String[] args) {
7        // assign superclass reference to superclass variable
8        CommissionEmployee commissionEmployee = new CommissionEmployee(
9           "Sue", "Jones", "222-22-2222", 10000, .06);
10
11       // assign subclass reference to subclass variable
12       BasePlusCommissionEmployee basePlusCommissionEmployee =
13          new BasePlusCommissionEmployee(
14          "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 5.)

```
15
16          // invoke toString on superclass object using superclass variable
17          System.out.printf("%s %s:%n%n%s%n%n",
18             "Call CommissionEmployee's toString with superclass reference ",
19             "to superclass object", commissionEmployee.toString());
20
21          // invoke toString on subclass object using subclass variable
22          System.out.printf("%s %s:%n%n%s%n%n",
23             "Call BasePlusCommissionEmployee's toString with subclass",
24             "reference to subclass object",
25             basePlusCommissionEmployee.toString());
26
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 5.)

```
27          // invoke toString on subclass object using superclass variable
28          CommissionEmployee commissionEmployee2 =
29             basePlusCommissionEmployee;
30          System.out.printf("%s %s:%n%n%s%n",
31             "Call BasePlusCommissionEmployee's toString with superclass",
32             "reference to subclass object", commissionEmployee2.toString());
33       }
34    }
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 3 of 5.)

```
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

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
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 4 of 5.)

```
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 5 of 5.)

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**.

10.3 Dynamic Binding (Cont.)

```java
class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Rectangle();

        shape1.draw(); // Output: Drawing a circle
        shape2.draw(); // Output: Drawing a rectangle
    }
}
```

We created two shape objects: shape1 of type Circle and shape2 of type Rectangle.

Although the reference type is Shape, the actual objects being referred to are Circle and Rectangle.

When we call the draw() method on shape1 and shape2, the JVM dynamically binds the appropriate version of the draw() method at runtime based on the actual type of the object.

This dynamic binding allows us to write code that works with a generic reference type (Shape), **but at runtime, the correct method implementation is determined based on the actual object type being referred to.**

This flexibility and polymorphic behaviour is one of the key advantages of dynamic binding in Java.

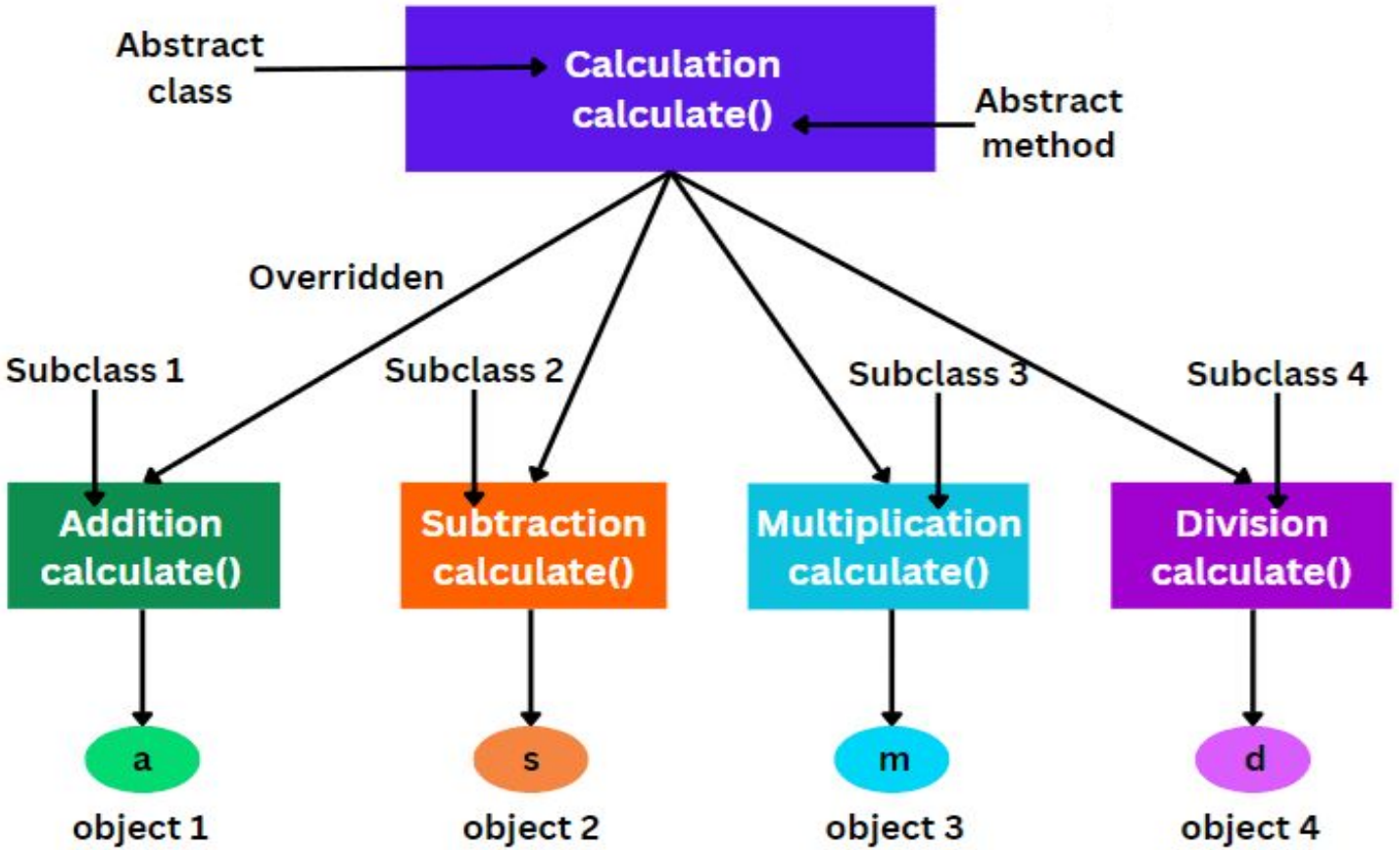
https://www.prepbytes.com/blog/java/dynamic-binding-in-java/#:~:text=Dynamic%20binding%20in%20Java%20refers,late%20binding%20or%20runtime%20polymorphism.

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




```python
# Import ABC
from abc import ABC, abstractmethod
# Define an abstract class named Calculation.
class Calculation(ABC):
    @abstractmethod
    def calculate(self, x, y):
        pass

# Define the subclasses with their own implementations.
class Addition(Calculation):
    def calculate(self, x, y):
        return x + y

class Subtraction(Calculation):
    def calculate(self, x, y):
        return x - y

class Multiplication(Calculation):
    def calculate(self, x, y):
        return x * y

class Division(Calculation):
    def calculate(self, x, y):
        return x / y

# Create an object of each subclass.
a = Addition()
s = Subtraction()
m = Multiplication()
d = Division()

# Call method using reference variable by passing arguments.
result1 = a.calculate(20, 30)
result2 = s.calculate(20, 10)
result3 = m.calculate(20, 40)
result4 = d.calculate(20, 5)

# Display the results.
print("Addition: ", result1)
print("Subtraction: ", result2)
print("Multiplication: ", result3)
print("Division: ", result4)
```

https://www.scientecheasy.com/2023/11/abstraction-in-python.html/

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.)

- Programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of subclass types.
  - You can write a method with a parameter of an abstract superclass type.
  - When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.
- Abstract classes sometimes constitute several levels of a hierarchy.

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 an instance method 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`.

Software Engineering Observation 10.4

An `abstract` class declares common attributes and behaviors (both `abstract` and concrete) of the classes in a class hierarchy. An `abstract` class typically contains one or more `abstract` methods that subclasses must override if they are to be concrete. The instance variables and concrete methods of an `abstract` class are subject to the normal rules of inheritance.

Common Programming Error 10.1

Attempting to instantiate an object of an abstract class is a compilation error.

Common Programming Error 10.2

Classes must be declared `abstract` if they declare `abstract` methods or if they inherit `abstract` methods and do not provide concrete implementations of them; otherwise, compilation errors occur.

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.
  - We'll 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.
  - Device drivers control all communication between the operating system and the devices.
  - A write message sent to a device-driver object is interpreted in the context of that driver and how it manipulates devices of a specific type.
  - The write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device.

10.4 Abstract Classes and Methods (Cont.)

- An object-oriented operating system might use an abstract superclass to provide an “interface” appropriate for all device drivers.
  - Subclasses are formed that all behave similarly.
  - The device-driver methods are declared as abstract methods in the abstract superclass.
  - The implementations of these abstract methods are provided in the subclasses that correspond to the specific types of device drivers.
- New devices are always being developed.
  - When you buy a new device, it comes with a device driver provided by the device vendor and is immediately operational after you connect it and install the driver.
- This is another elegant example of how polymorphism makes systems extensible.

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 you 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.

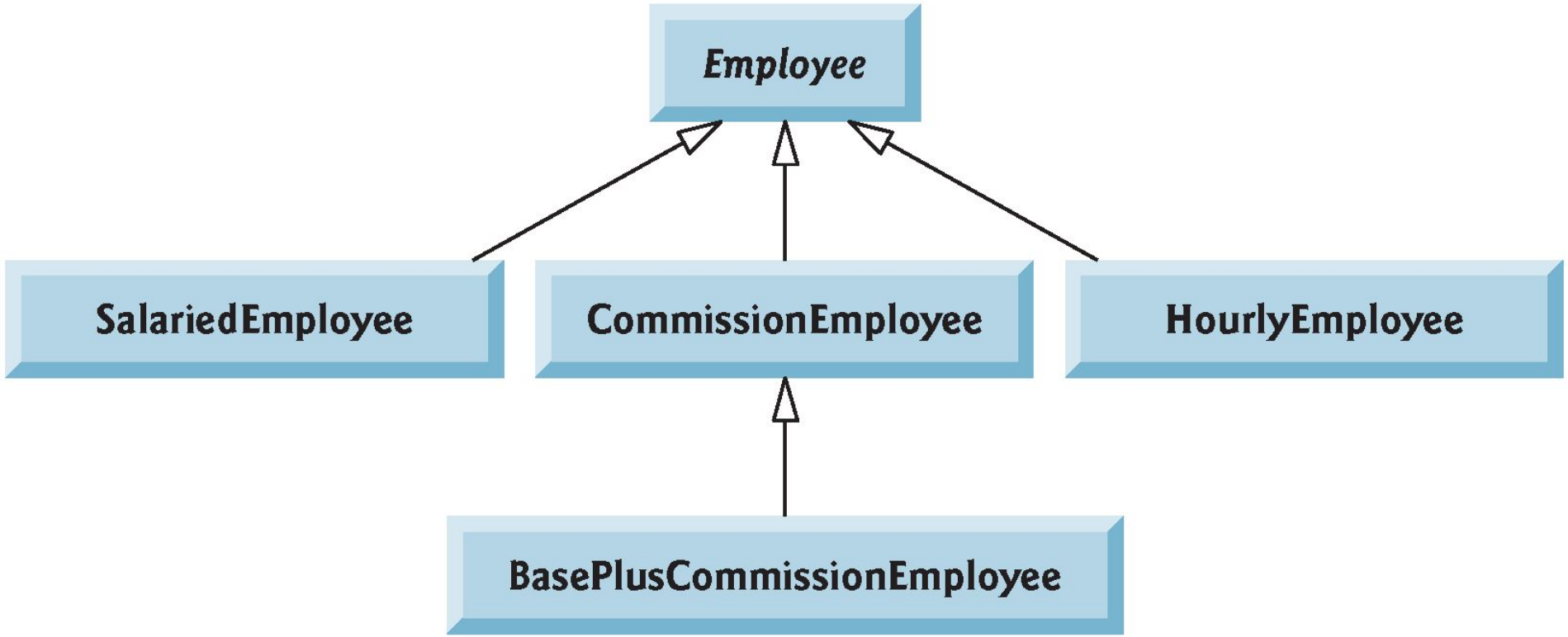


Fig. 10.2 | `Employee` hierarchy UML class diagram.

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 `Employees` 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.

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8
9     // constructor
10    public Employee(String firstName, String lastName,
11       String socialSecurityNumber) {
12       this.firstName = firstName;
13       this.lastName = lastName;
14       this.socialSecurityNumber = socialSecurityNumber;
15    }
16
```

Fig. 10.4 | Employee abstract superclass. (Part 1 of 2.)

```
17      // return first name
18      public String getFirstName() {return firstName;}
19
20      // return last name
21      public String getLastName() {return lastName;}
22
23      // return social security number
24      public String getSocialSecurityNumber() {return socialSecurityNumber;}
25
26      // return String representation of Employee object
27      @Override
28      public String toString() {
29         return String.format("%s %s%nsocial security number: %s",
30            getFirstName(), getLastName(), getSocialSecurityNumber());
31      }
32
33      // abstract method must be overridden by concrete subclasses
34      public abstract double earnings(); // no implementation here
35   }
```

Fig. 10.4 | `Employee` abstract superclass. (Part 2 of 2.)

10.5.2 Concrete Subclass `SalariedEmployee`

Error-Prevention Tip 10.1

We've said that you should not call a class's instance methods from its constructors—you can call `static` class methods and make the required call to one of the superclass's constructors. If you follow this advice, you'll avoid the problem of calling the class's overridable methods either directly or indirectly, which can lead to runtime errors. See Section 10.8 for additional details.

```
1  // Fig. 10.5: SalariedEmployee.java
2  // SalariedEmployee concrete class extends abstract class Employee.
3
4  public class SalariedEmployee extends Employee {
5     private double weeklySalary;
6
7     // constructor
8     public SalariedEmployee(String firstName, String lastName,
9        String socialSecurityNumber, double weeklySalary) {
10       super(firstName, lastName, socialSecurityNumber);
11
12       if (weeklySalary < 0.0) {
13          throw new IllegalArgumentException(
14             "Weekly salary must be >= 0.0");
15       }
16
17       this.weeklySalary = weeklySalary;
18    }
```

Fig. 10.5 | SalariedEmployee concrete class extends abstract class Employee. (Part 1 of 3.)

```
19
20      // set salary
21      public void setWeeklySalary(double weeklySalary) {
22         if (weeklySalary < 0.0) {
23            throw new IllegalArgumentException(
24               "Weekly salary must be >= 0.0");
25         }
26
27         this.weeklySalary = weeklySalary;
28      }
29
30      // return salary
31      public double getWeeklySalary() {return weeklySalary;}
32
```

Fig. 10.5 | `SalariedEmployee` concrete class extends `abstract` class `Employee`. (Part 2 of 3.)

```
33      // calculate earnings; override abstract method earnings in Employee
34      @Override
35      public double earnings() {return getWeeklySalary();}
36
37      // return String representation of SalariedEmployee object
38      @Override
39      public String toString() {
40         return String.format("salaried employee: %s%n%s: $%,.2f",
41            super.toString(), "weekly salary", getWeeklySalary());
42      }
43   }
```

Fig. 10.5 | `SalariedEmployee` concrete class extends `abstract` class `Employee`. (Part 3 of 3.)

10.5.3 Concrete Subclass HourlyEmployee

```
1  // Fig. 10.6: HourlyEmployee.java
2  // HourlyEmployee class extends Employee.
3
4  public class HourlyEmployee extends Employee {
5     private double wage; // wage per hour
6     private double hours; // hours worked for week
7
```

Fig. 10.6 | HourlyEmployee class extends Employee. (Part 1 of 5.)

```
 8      // constructor
 9      public HourlyEmployee(String firstName, String lastName,
10         String socialSecurityNumber, double wage, double hours) {
11         super(firstName, lastName, socialSecurityNumber);
12
13         if (wage < 0.0) { // validate wage
14            throw new IllegalArgumentException("Hourly wage must be >= 0.0");
15         }
16
17         if ((hours < 0.0) || (hours > 168.0)) { // validate hours
18            throw new IllegalArgumentException(
19               "Hours worked must be >= 0.0 and <= 168.0");
20         }
21
22         this.wage = wage;
23         this.hours = hours;
24      }
```

Fig. 10.6 | HourlyEmployee class extends Employee. (Part 2 of 5.)

```
25
26      // set wage
27      public void setWage(double wage) {
28         if (wage < 0.0) { // validate wage
29            throw new IllegalArgumentException("Hourly wage must be >= 0.0");
30         }
31
32         this.wage = wage;
33      }
34
35      // return wage
36      public double getWage() {return wage;}
37
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 3 of 5.)

```
38      // set hours worked
39      public void setHours(double hours) {
40         if ((hours < 0.0) || (hours > 168.0)) { // validate hours
41            throw new IllegalArgumentException(
42               "Hours worked must be >= 0.0 and <= 168.0");
43         }
44
45         this.hours = hours;
46      }
47
48      // return hours worked
49      public double getHours() {return hours;}
50
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 4 of 5.)

```
51      // calculate earnings; override abstract method earnings in Employee
52      @Override
53      public double earnings() {
54         if (getHours() <= 40) { // no overtime
55            return getWage() * getHours();
56         }
57         else {
58            return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
59         }
60      }
61
62      // return String representation of HourlyEmployee object
63      @Override
64      public String toString() {
65         return String.format("hourly employee: %s%n%s: $%,.2f; %s: %,.2f",
66            super.toString(), "hourly wage", getWage(),
67            "hours worked", getHours());
68      }
69   }
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 5 of 5.)

10.5.4 Concrete Subclass CommissionEmployee

```
1  // Fig. 10.7: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3
4  public class CommissionEmployee extends Employee {
5     private double grossSales; // gross weekly sales
6     private double commissionRate; // commission percentage
7
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 1 of 5.)

```
 8      // constructor
 9      public CommissionEmployee(String firstName, String lastName,
10         String socialSecurityNumber, double grossSales,
11         double commissionRate) {
12         super(firstName, lastName, socialSecurityNumber);
13
14         if (commissionRate <= 0.0 || commissionRate >= 1.0) { // validate
15            throw new IllegalArgumentException(
16               "Commission rate must be > 0.0 and < 1.0");
17         }
18
19         if (grossSales < 0.0) { // validate
20            throw new IllegalArgumentException("Gross sales must be >= 0.0");
21         }
22
23         this.grossSales = grossSales;
24         this.commissionRate = commissionRate;
25      }
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 2 of 5.)

```
26
27      // set gross sales amount
28      public void setGrossSales(double grossSales) {
29         if (grossSales < 0.0) { // validate
30            throw new IllegalArgumentException("Gross sales must be >= 0.0");
31         }
32
33         this.grossSales = grossSales;
34      }
35
36      // return gross sales amount
37      public double getGrossSales() {return grossSales;}
38
```

Fig. 10.7 | `CommissionEmployee` class extends `Employee`. (Part 3 of 5.)

```
39      // set commission rate
40      public void setCommissionRate(double commissionRate) {
41         if (commissionRate <= 0.0 || commissionRate >= 1.0) { // validate
42            throw new IllegalArgumentException(
43               "Commission rate must be > 0.0 and < 1.0");
44         }
45
46         this.commissionRate = commissionRate;
47      }
48
49      // return commission rate
50      public double getCommissionRate() {return commissionRate;}
51
```

Fig. 10.7 | `CommissionEmployee` class extends `Employee`. (Part 4 of 5.)

```
52      // calculate earnings; override abstract method earnings in Employee
53      @Override
54      public double earnings() {
55         return getCommissionRate() * getGrossSales();
56      }
57
58      // return String representation of CommissionEmployee object
59      @Override
60      public String toString() {
61         return String.format("%s: %s%n%s: $%,.2f; %s: %.2f",
62            "commission employee", super.toString(),
63            "gross sales", getGrossSales(),
64            "commission rate", getCommissionRate());
65      }
66   }
```

Fig. 10.7 | `CommissionEmployee` class extends `Employee`. (Part 5 of 5.)

10.5.5 Indirect Concrete Subclass BasePlusCommissionEmployee

```
1  // Fig. 10.8: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class extends CommissionEmployee.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee {
5     private double baseSalary; // base salary per week
6
7     // constructor
8     public BasePlusCommissionEmployee(String firstName, String lastName,
9        String socialSecurityNumber, double grossSales,
10       double commissionRate, double baseSalary) {
11       super(firstName, lastName, socialSecurityNumber,
12          grossSales, commissionRate);
13
14       if (baseSalary < 0.0) { // validate baseSalary
15          throw new IllegalArgumentException("Base salary must be >= 0.0");
16       }
17
18       this.baseSalary = baseSalary;
19    }
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee. (Part 1 of 3.)

```
20
21      // set base salary
22      public void setBaseSalary(double baseSalary) {
23         if (baseSalary < 0.0) { // validate baseSalary
24            throw new IllegalArgumentException("Base salary must be >= 0.0");
25         }
26
27         this.baseSalary = baseSalary;
28      }
29
30      // return base salary
31      public double getBaseSalary() {return baseSalary;}
32
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee. (Part 2 of 3.)

```
33      // calculate earnings; override method earnings in CommissionEmployee
34      @Override
35      public double earnings() {return getBaseSalary() + super.earnings();}
36
37      // return String representation of BasePlusCommissionEmployee object
38      @Override
39      public String toString() {
40         return String.format("%s %s; %s: $%,.2f",
41            "base-salaried", super.toString(),
42            "base salary", getBaseSalary());
43      }
44   }
```

Fig. 10.8 | `BasePlusCommissionEmployee` class extends `CommissionEmployee`. (Part 3 of 3.)

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.

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest {
5     public static void main(String[] args) {
6        // create subclass objects
7        SalariedEmployee salariedEmployee =
8           new SalariedEmployee("John", "Smith", "111-11-1111", 800.00);
9        HourlyEmployee hourlyEmployee =
10          new HourlyEmployee("Karen", "Price", "222-22-2222", 16.75, 40);
11       CommissionEmployee commissionEmployee =
12          new CommissionEmployee(
13          "Sue", "Jones", "333-33-3333", 10000, .06);
14       BasePlusCommissionEmployee basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee(
16          "Bob", "Lewis", "444-44-4444", 5000, .04, 300);
17
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 1 of 8.)

```
18          System.out.println("Employees processed individually:");
19
20          System.out.printf("%n%s%n%s: $%,.2f%n%n",
21             salariedEmployee, "earned", salariedEmployee.earnings());
22          System.out.printf("%s%n%s: $%,.2f%n%n",
23             hourlyEmployee, "earned", hourlyEmployee.earnings());
24          System.out.printf("%s%n%s: $%,.2f%n%n",
25             commissionEmployee, "earned", commissionEmployee.earnings());
26          System.out.printf("%s%n%s: $%,.2f%n%n",
27             basePlusCommissionEmployee,
28             "earned", basePlusCommissionEmployee.earnings());
29
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 2 of 8.)

```
30          // create four-element Employee array
31          Employee[] employees = new Employee[4];
32
33          // initialize array with Employees
34          employees[0] = salariedEmployee;
35          employees[1] = hourlyEmployee;
36          employees[2] = commissionEmployee;
37          employees[3] = basePlusCommissionEmployee;
38
39          System.out.printf("Employees processed polymorphically:%n%n");
40
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 3 of 8.)

```
41          // generically process each element in array employees
42          for (Employee currentEmployee : employees) {
43             System.out.println(currentEmployee); // invokes toString
44
45             // determine whether element is a BasePlusCommissionEmployee
46             if (currentEmployee instanceof BasePlusCommissionEmployee) {
47                // downcast Employee reference to
48                // BasePlusCommissionEmployee reference
49                BasePlusCommissionEmployee employee =
50                   (BasePlusCommissionEmployee) currentEmployee;
51
52                employee.setBaseSalary(1.10 * employee.getBaseSalary());
53
54                System.out.printf(
55                   "new base salary with 10%% increase is: $%,.2f%n",
56                   employee.getBaseSalary());
57             }
58
59             System.out.printf(
60                "earned $%,.2f%n%n", currentEmployee.earnings());
61          }
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 4 of 8.)

```
62
63          // get type name of each object in employees array
64          for (int j = 0; j < employees.length; j++) {
65             System.out.printf("Employee %d is a %s%n", j,
66                employees[j].getClass().getName());
67          }
68       }
69    }
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 5 of 8.)

```
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 6 of 8.)

```
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 7 of 8.)

```
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 8 of 8.)

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.

Common Programming Error 10.3

Assigning a superclass variable to a subclass variable is a compilation error.

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 *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.

Software Engineering Observation 10.5

Although the actual method that's called depends on the runtime type of the object to which a variable refers, a variable can be used to invoke only those methods that are members of that variable's type, which the compiler verifies.

10.6 Summary of the Allowed Assignments Between Superclass and Subclass Variables

- There are three proper 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 straightfor-ward.
- Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an object of its superclass.*
  - The superclass variable can be used to refer only to superclass members.
  - If this code refers to subclass-only mem-bers through the superclass variable, the compiler reports errors.

10.7 `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.

10.7 `final` Methods and Classes (Cont.)

- A `final` class cannot be extended to create a subclass.
  - All methods in a `final` class are implicitly `final`.
- Class `String` is an example of a `final` class.
  - If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected.
  - Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions.

Common Programming Error 10.5

Attempting to declare a subclass of a `final` class is a compilation error.

Software Engineering Observation 10.6

In the Java API, the vast majority of classes are not declared `final`. This enables inheritance and polymorphism. However, in some cases, it's important to declare classes `final`—typically for security reasons. Also, unless you carefully design a class for extension, you should declare the class as `final` to avoid (often subtle) errors.

Software Engineering Observation 10.7

Though `final` classes cannot be extended, you can reuse them via composition.

10.8 A Deeper Explanation of Issues with Calling Methods from Constructors

- Do not call overridable methods from constructors.
- When creating a *subclass* object, this could lead to an overridden method being called before the *subclass* object is fully initialized.
- Recall that when you construct a *subclass* object, its constructor first calls one of the direct *superclass's* constructors.
- If the *superclass* constructor calls an overridable method, the *subclass's* version of that method will be called by the *superclass* constructor—before the *subclass* constructor's body has a chance to execute.
- This could lead to subtle, difficult-to-detect errors if the *subclass* method that was called depends on initialization that has not yet been performed in the *subclass* constructor's body.
- It's acceptable to call a `static` method from a constructor.

10.8 A Deeper Explanation of Issues with Calling Methods from Constructors

- Let's assume that a constructor and a *set* method perform the same validation for a particular instance variable. How should you handle the common code?
  - If it's brief, you can duplicate it in the constructor and the *set* method
  - For lengthier validation, define a `static` validation method—typically a `private static` helper method—then call it from the constructor and from the *set* method. It's acceptable to call a `static` method from a constructor, because `static` methods are not overridable.
- It's also acceptable for a constructor to call a `final` instance method, provided that the method does not directly or indirectly call any overridable instance methods.

10.9 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.

10.9 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.

10.9 Creating and Using Interfaces (Cont.)

- 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).
  - The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.

10.9 Creating and Using Interfaces (Cont.)

- Example: 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*
  - e.g., 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.

10.9 Creating and Using Interfaces (Cont.)

- A Java interface describes a set of methods that can be called on an object.
- 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`.

10.9 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

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`.

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 should be used in place of an `abstract` class when there is no default implementation to inherit—that is, no fields and no concrete method implementations.
- Like `public abstract` classes, interfaces are typically `public` types.
- 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).

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 `Invoices` and `Employees`.

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.

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 class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface.
- A subclass inherits its superclass's realization relationships.

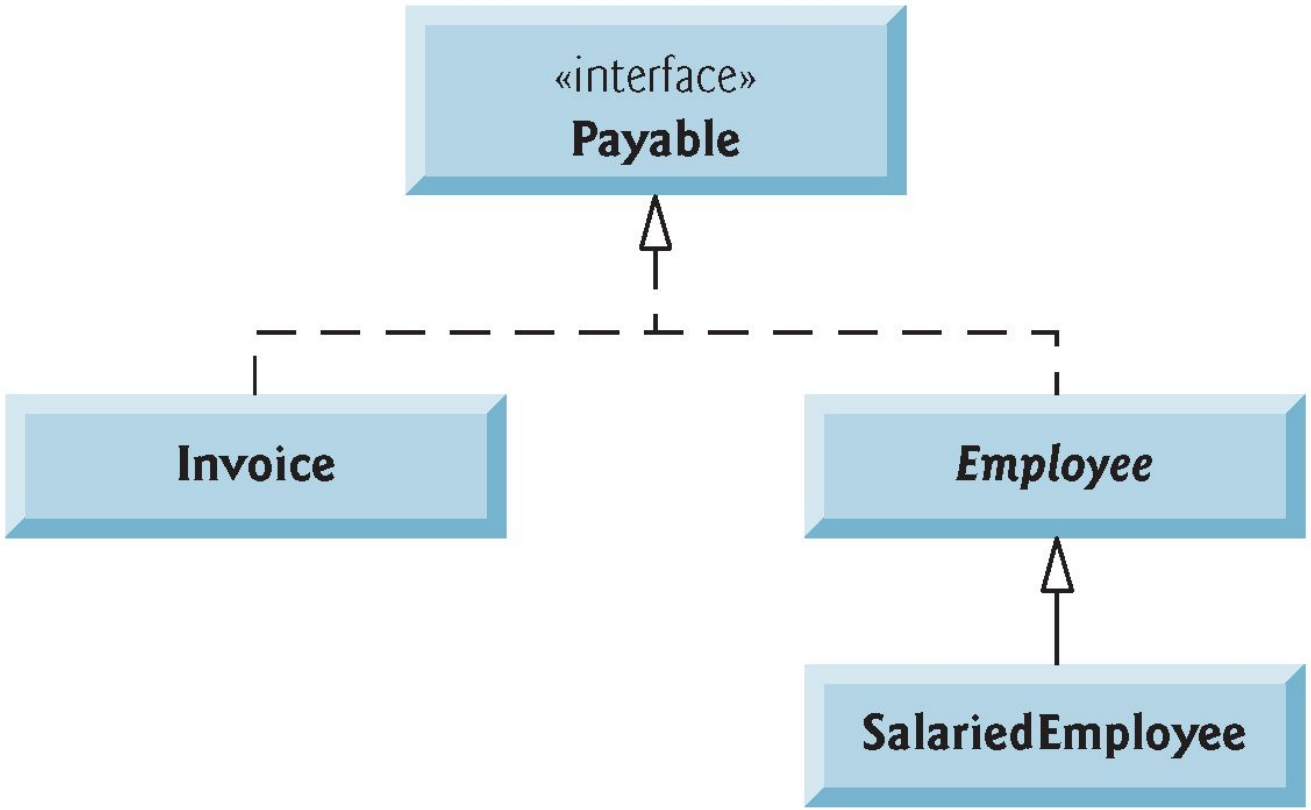


Fig. 10.10 | `Payable` hierarchy UML class diagram.

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.

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     public abstract double getPaymentAmount(); // no implementation
6  }
```

Fig. 10.11 | `Payable` interface declaration.

10.9.3 Class `Invoice`

- Class `Invoice` (Fig. 19.12) represents a simple invoice that contains billing information for only one kind of part
- 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, 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     private final String partNumber;
6     private final String partDescription;
7     private final int quantity;
8     private final double pricePerItem;
9
```

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

```
10      // constructor
11      public Invoice(String partNumber, String partDescription, int quantity,
12         double pricePerItem) {
13         if (quantity < 0) { // validate quantity
14            throw new IllegalArgumentException("Quantity must be >= 0");
15         }
16
17         if (pricePerItem < 0.0) { // validate pricePerItem
18            throw new IllegalArgumentException(
19               "Price per item must be >= 0");
20         }
21
22         this.quantity = quantity;
23         this.partNumber = partNumber;
24         this.partDescription = partDescription;
25         this.pricePerItem = pricePerItem;
26      }
```

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

```
27
28      // get part number
29      public String getPartNumber() {return partNumber;}
30
31      // get description
32      public String getPartDescription() {return partDescription;}
33
34      // get quantity
35      public int getQuantity() {return quantity;}
36
37      // get price per item
38      public double getPricePerItem() {return pricePerItem;}
39
```

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

```
40      // return String representation of Invoice object
41      @Override
42      public String toString() {
43         return String.format("%s: %n%s: %s (%s) %n%s: %d %n%s: $%,.2f",
44            "invoice", "part number", getPartNumber(), getPartDescription(),
45            "quantity", getQuantity(), "price per item", getPricePerItem());
46      }
47
48      // method required to carry out contract with interface Payable
49      @Override
50      public double getPaymentAmount() {
51         return getQuantity() * getPricePerItem(); // calculate total cost
52      }
53   }
```

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

Software Engineering Observation 10.9

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`

- We now modify class `Employee` to implement interface `Payable`. This class declaration is identical to that of with two exceptions:
  - Line 4 of indicates that class `Employee` now implements `Payable`.
  - Line 38 implements interface `Payable`'s `getPaymentAmount` method.
- `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 Payable.
3
4  public abstract class Employee implements Payable {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8
9     // constructor
10    public Employee(String firstName, String lastName,
11       String socialSecurityNumber) {
12       this.firstName = firstName;
13       this.lastName = lastName;
14       this.socialSecurityNumber = socialSecurityNumber;
15    }
16
```

Fig. 10.13 | `Employee` `abstract` superclass that implements `Payable`. (Part 1 of 3.)

```
17      // return first name
18      public String getFirstName() {return firstName;}
19
20      // return last name
21      public String getLastName() {return lastName;}
22
23      // return social security number
24      public String getSocialSecurityNumber() {return socialSecurityNumber;}
25
26      // return String representation of Employee object
27      @Override
28      public String toString() {
29         return String.format("%s %s%nsocial security number: %s",
30            getFirstName(), getLastName(), getSocialSecurityNumber());
31      }
32
```

Fig. 10.13 | `Employee` `abstract` superclass that implements `Payable`. (Part 2 of 3.)

```
33      // abstract method must be overridden by concrete subclasses
34      public abstract double earnings(); // no implementation here
35
36      // implementing getPaymentAmount here enables the entire Employee
37      // class hierarchy to be used in an app that processes Payables
38      public double getPaymentAmount() {return earnings();}
39  }
```

Fig. 10.13 | `Employee` `abstract` superclass that implements `Payable`. (Part 3 of 3.)

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 `Invoices` and `Employees` *polymorphically* in a single application.
- 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*

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

- 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 getPayment-Amount methods.

```
1  // Fig. 10.14: PayableInterfaceTest.java
2  // Payable interface test program processing Invoices and
3  // Employees polymorphically.
4  public class PayableInterfaceTest {
5     public static void main(String[] args) {
6        // create four-element Payable array
7        Payable[] payableObjects = new Payable[] {
8           new Invoice("01234", "seat", 2, 375.00),
9           new Invoice("56789", "tire", 4, 79.95),
10          new SalariedEmployee("John", "Smith", "111-11-1111", 800.00),
11          new SalariedEmployee("Lisa", "Barnes", "888-88-8888", 1200.00)
12       };
13
```

Fig. 10.14 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 1 of 3.)

```
14          System.out.println(
15             "Invoices and Employees processed polymorphically:");
16
17          // generically process each element in array payableObjects
18          for (Payable currentPayable : payableObjects) {
19             // output currentPayable and its appropriate payment amount
20             System.out.printf("%n%s %npayment due: $%,.2f%n",
21                currentPayable.toString(), // could invoke implicitly
22                currentPayable.getPaymentAmount());
23          }
24       }
25    }
```

Fig. 10.14 | `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. 10.14 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 3 of 3.)

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 `ArrayList`. We use `Comparable` in Chapter 16, Generic Collections, and Chapter 20, Generic Classes and Methods: A Deeper Look.

Fig. 10.15 | Common interfaces of the Java API. (Part 1 of 4.)

Interface	Description
`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.

Fig. 10.15 | Common interfaces of the Java API. (Part 2 of 4.)

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, 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.

Fig. 10.15 | Common interfaces of the Java API. (Part 3 of 4.)

Interface	Description
`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. (Part 4 of 4.)

10.10 Java SE 8 Interface Enhancements

- This section introduces interface features that were added in
- Java SE
- 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
- This gives you more flexibility in designing your classes, because a class can implement many interfaces, but can extend only one superclass

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—also called SAM (Single Abstract Method) interfaces
- 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.

Software Engineering Observation 10.13

Java SE 8 `default` methods enable you to evolve existing interfaces by adding new methods to those interfaces without breaking code that uses them.

10.11 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`.

10.12 private Constructors

Sometimes it's useful to declare one or more of a class's constructors as private.

Preventing Object Instantiation

- You can prevent client code from creating objects of a class by making the class's constructors private
- 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`

10.12 private Constructors (cont.)

Sharing Initialization Code in Constructors

- One common use of a `private` constructor is sharing initialization code among a class's other constructors
- You can use delegating constructors (introduced in Fig. 8.5) to call the `private` constructor that contains the shared initialization code

10.12 private Constructors (cont.)

Factory Methods

- Another common use of `private` constructors is to force client code to use so-called "factory methods" to create objects
- A **factory method** is a `public static` method that creates and initializes an object of a specified type (possibly of the same class), then returns a reference to it
- A key benefit of this architecture is that the method's return type can be an interface or a superclass (either abstract or concrete)

10.13 Program to an Interface, Not an Implementation

- Recall that Java does not allow a class to inherit from more than one superclass
- With interface inheritance, a class implements an interface describing various abstract methods that the new class must provide
- The new class also may inherit some method implementations (allowed in interfaces as of Java SE 8), but no instance variables
- Recall that Java allows a class to implement multiple interfaces in addition to extending one class
- An interface also may extend one or more other interfaces.

10.13.1 Implementation Inheritance Is Best for Small Numbers of Tightly Coupled Classes

- Implementation inheritance is primarily used to declare closely related classes
  - many of the same instance variables and method implementations
- Every subclass object has the *is-a* relationship with the superclass
  - anywhere a superclass object is expected, a subclass object may be provided
- Classes declared with implementation inheritance are tightly coupled
  - you define the common instance variables and methods once in a superclass, then inherit them into subclasses
- Changes to a superclass directly affect all corresponding subclasses
- When you use a superclass variable, only a superclass object or one of its subclass objects may be assigned to the variable.

10.13.1 Implementation Inheritance Is Best for Small Numbers of Tightly Coupled Classes (cont.)

- A key disadvantage of implementation inheritance is that the tight coupling among the classes can make it difficult to modify the hierarchy
- As we mentioned in Chapter 9, small inheritance hierarchies under the control of one person tend to be more manageable than large ones maintained by many people
- This is true even with the tight coupling associated with implementation inheritance

10.13.2 Interface Inheritance Is Best for Flexibility

- Interface inheritance often requires more work than implementation inheritance, because you must provide implementations of the interface's abstract methods
  - even if those implementations are similar or identical among classes
- Gives you additional flexibility by eliminating the tight coupling between classes
- When you use a variable of an interface type, you can assign it an object of *any* type that implements the interface directly or indirectly
  - Allows you to add new types to your code easily and to replace existing objects with objects of new and improved implementation classes.
  - Device drivers are a good example of how interfaces enable systems to be modified easily

Software Engineering Observation 10.14

Java SE 8's and Java SE 9's `interface` enhancements (Sections 10.10–10.11)—which allow `interfaces` to contain `public` and `private` instance methods and `static` methods with implementations—make programming with `interfaces` appropriate for almost all cases in which you would have used `abstract` classes previously. With the exception of fields, you get all the benefits that classes provide, plus classes can implement any number of interfaces but can extend only one class (`abstract` or concrete).

Software Engineering Observation 10.15

Just as superclasses can change, so can interfaces. If the signature of an interface method changes, all corresponding classes would require modification. Experience has shown that interfaces change much less frequently than implementations.

10.13.3 Rethinking the Employee Hierarchy

- Let's reconsider the `Employee` hierarchy with composition and an interface
- Can say that each type of employee in the hierarchy is an `Employee` that *has a* `CompensationModel`
- Can declare `CompensationModel` as an interface with an `abstract` `earnings` method, then declare implementations of `CompensationModel` that specify the various ways in which an Employee gets paid

10.13.3 Rethinking the Employee Hierarchy

- A `SalariedCompensationModel` would contain a `weeklySalary` instance variable and would implement `earnings` to return the `weeklySalary`.
- An `HourlyCompensationModel` would contain `wage` and `hours` instance variables and would implement `earnings` based on the number of hours worked, with `1.5 * wage` for any hours over 40.
- A `CommissionCompensationModel` would contain `grossSales` and `commissionRate` instance variables and would implement `earnings` to return `grossSales * commissionRate`.
- A `BasePlusCommissionCompensationModel` would contain instance variables `grossSales`, `commissionRate` and `baseSalary` and would implement `earnings` to return `baseSalary + grossSales * commissionRate`

10.13.3 Rethinking the Employee Hierarchy

- Each `Employee` object you create can then be initialized with an object of the appropriate `CompensationModel` implementation
- Class `Employee`'s `earnings` method would simply use the class's composed `CompensationModel` instance variable to call the `earnings` method of the corresponding `CompensationModel` object

10.13.3 Rethinking the Employee Hierarchy

Flexibility if Compensation Models Change

- Declaring the `CompensationModels` as separate classes that implement the same interface provides flexibility for future changes
- Assume `Employees` who are paid by commission based on gross sales should get an extra 10% commission, but those who have a base salary should not
- In the original `Employee` hierarchy, making this change to class `CommissionEmployee`'s `earnings` method directly affects how `BasePlusCommissionEmployees` are paid, because `BasePlusCommissionEmployee`'s `earnings` method calls `CommissionEmployee`'s `earnings` method.
- But changing the `CommissionCompensationModel`'s `earnings` implementation does not affect `BasePlusCommissionCompensationModel`, because these classes are not tightly coupled by inheritance

10.13.3 Rethinking the Employee Hierarchy

Flexibility if Employees Are Promoted

- Interface-based composition is more flexible than Section 10.5's class hierarchy if an `Employee` gets promoted
- Class `Employee` can provide a `setCompensationModel` method that receives a `CompensationModel` and assigns it to the `Employee`'s composed `CompensationModel` variable
- When an `Employee` gets promoted, you'd simply call `setCompensationModel` to replace the `Employee`'s existing `CompensationModel` object with an appropriate new one
- To promote an employee using 's `Employee` hierarchy, you'd need to change the employee's type by creating a new object of the appropriate class and moving data from the old object into the new one.
- Exercise 10.18 asks you to reimplement Exercise 9.16 using interface `CompensationModel` as described in this section.

10.13.3 Rethinking the Employee Hierarchy

Flexibility if Employees Acquire New Capabilities

- Using composition and interfaces also is more flexible than Section 10.5's class hierarchy for enhancing class `Employee`
- Let's assume we decide to support retirement plans (such as 401Ks and IRAs). We could say that every `Employee` *has a* `RetirementPlan` and define interface `RetirementPlan` with a `makeRetirementDeposit` method
- Then, we can provide appropriate implementations for various retirement-plan types

10.14 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism

- Shape classes have many similarities.
- Using inheritance, we can “factor out” the common features from all three classes and place them in a single shape *superclass*.
- Then, using variables of the superclass type, we can manipulate objects of all three shape objects *polymorphically*.
- Removing the redundancy in the code will result in a smaller, more flexible program that is easier to maintain.

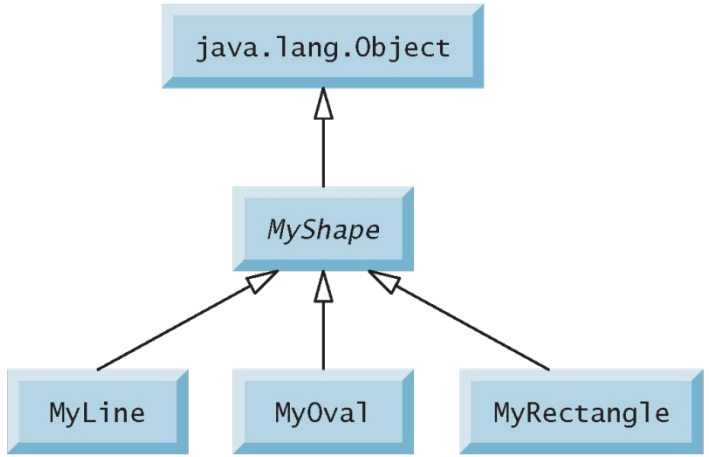


Fig. 10.17 | MyShape hierarchy.

10.14 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism (Cont.)

- Class `MyBoundedShape` can be used to factor out the common features of classes `MyOval` and `MyRectangle`.

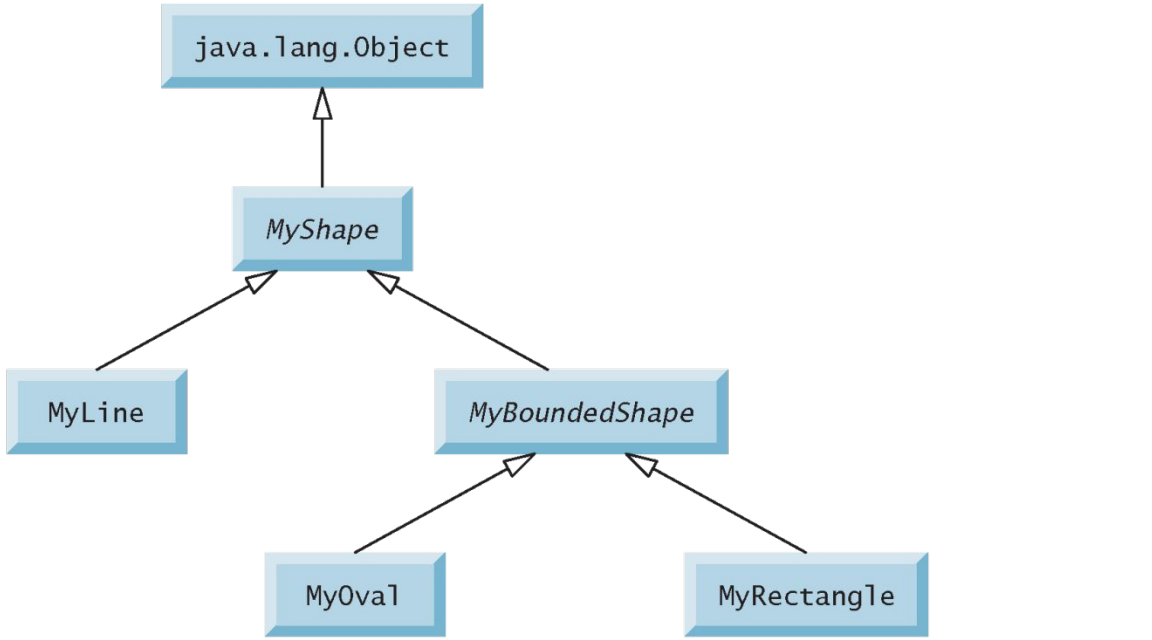


Fig. 10.18 | MyShape hierarchy with MyBoundedShape.