JAVA Programming 2014-2
Java Programming Part-II Object-Oriented Programming

1. Inheritance

Introduction to Inheritance

- A form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities.
- Can save time during program development by basing new classes on existing proven and debugged high-quality software.
- Increases the likelihood that a system will be implemented and maintained effectively.
- When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class.
    - Existing class is the superclass
    - New class is the subclass
- Each subclass can be a superclass of future subclasses.
- A subclass can add its own fields and methods.
- A subclass is more specific than its superclass and represents a more specialized group of objects.
- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
- This is why inheritance is sometimes referred to as specialization.
- The direct superclass is the superclass from which the subclass explicitly inherits.
- An indirect superclass is any class above the direct superclass in the class hierarchy.
- The Java class hierarchy begins with class Object (in package java.lang)
- Every class in Java directly or indirectly extends (or "inherits from") Object.
- Java supports only single inheritance, in which each class is derived from exactly one direct superclass.
- We distinguish between the is-a relationship and the has-a relationship
- Is-a represents inheritance, In an is-a relationship, an object of a subclass can also be treated as an object of its superclass
- Has-a represents composition, In a has-a relationship, an object contains as members references to other objects

Superclasses and Subclasses

- Figure 9.1 lists several simple examples of superclasses and subclasses
- Superclasses tend to be "more general" and subclasses "more specific."
- Because every subclass object is an object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is typically larger than the set of objects represented by any of its subclasses.
-

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Fig. 9.1 | Inheritance examples.

- A superclass exists in a hierarchical relationship with its subclasses.
- Fig. 9.2 shows a sample university community class hierarchy. Also called an inheritance hierarchy.
- Each arrow in the hierarchy represents an is-a relationship.
- Follow the arrows upward in the class hierarchy
- an Employee is a CommunityMember"
- "a Teacher is a Faculty member."
- CommunityMember is the direct superclass of Employee, Student and Alumnus and is an indirect superclass of all the other classes in the diagram.
- Starting from the bottom, you can follow the arrows and apply the is-a relationship up to the topmost superclass.

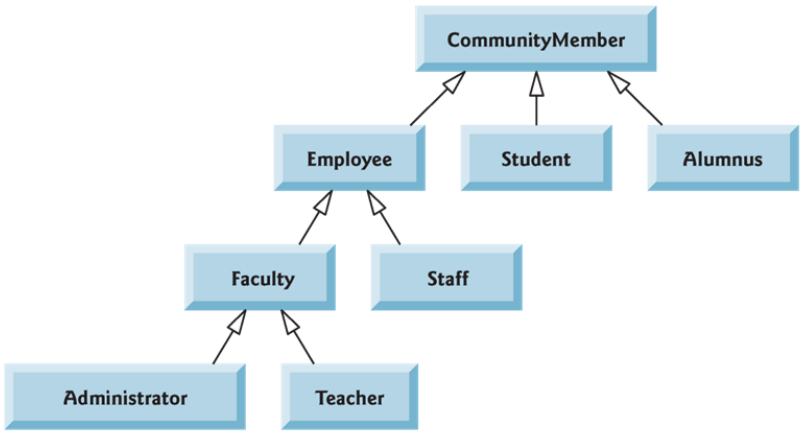

Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`.

- Fig. 9.3 shows a Shape inheritance hierarchy.
- Van follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several is-a relationships.
- A Triangle is a TwoDimensionalShape and is a Shape
- A Sphere is a ThreeDimensionalShape and is a Shape.

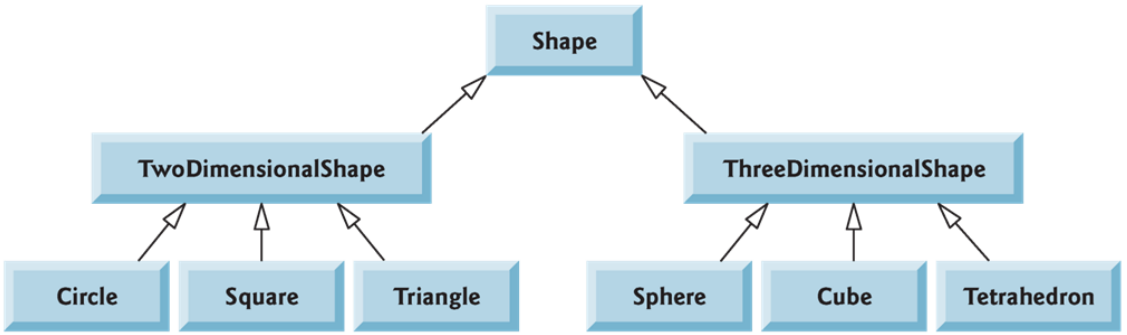

Fig. 9.3 | Inheritance hierarchy for Shapes.

- Inheritance issue
  - A subclass can inherit methods that it does not need or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.
  - The subclass can override (redefine) the superclass method with an appropriate implementation.

protected Members

- A class's public members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A class's private members are accessible only within the class itself.
- protected access is an intermediate level of access between public and private.
- A superclass's protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package
- protected members also have package access.
- All public and protected superclass members retain their original access modifier when they become members of the subclass.
- A superclass's private members are hidden in its subclasses
- They can be accessed only through the public or protected methods inherited from the superclass
- Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names.
- When a subclass method overrides an inherited superclass method, the superclass method can be accessed from the subclass by preceding the superclass method name with keyword super and a dot (.) separator.

Case Study (two types of Employees)

- Inheritance hierarchy containing types of employees in a company's payroll application
- Commission employees are paid a percentage of their sales
- Base-salaried commission employees receive a base salary plus a percentage of their sales.

Case Study (scenario-1- without using inheritance between Employees)

Creating and Using a `CommissionEmployee` Class

```
1  // Fig. 9.4: CommissionEmployee.java
2  // CommissionEmployee class represents an employee paid a
3  // percentage of gross sales.
4  public class CommissionEmployee extends Object
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs here
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate and store gross sales
21       setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 6.)

```
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first; // should validate
28    } // end method setFirstName
29
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last; // should validate
40    } // end method setLastName
41
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 6.)

```
42      // return last name
43      public String getLastName()
44      {
45         return lastName;
46      } // end method getLastName
47
48      // set social security number
49      public void setSocialSecurityNumber( String ssn )
50      {
51         socialSecurityNumber = ssn; // should validate
52      } // end method setSocialSecurityNumber
53
54      // return social security number
55      public String getSocialSecurityNumber()
56      {
57         return socialSecurityNumber;
58      } // end method getSocialSecurityNumber
59
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 3 of 6.)

```
60      // set gross sales amount
61      public void setGrossSales( double sales )
62      {
63         if ( sales >= 0.0 )
64            grossSales = sales;
65         else
66            throw new IllegalArgumentException(
67               "Gross sales must be >= 0.0" );
68      } // end method setGrossSales
69
70      // return gross sales amount
71      public double getGrossSales()
72      {
73         return grossSales;
74      } // end method getGrossSales
75
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 4 of 6.)

```
76      // set commission rate
77      public void setCommissionRate( double rate )
78      {
79         if ( rate > 0.0 && rate < 1.0 )
80            commissionRate = rate;
81         else
82            throw new IllegalArgumentException(
83               "Commission rate must be > 0.0 and < 1.0" );
84      } // end method setCommissionRate
85
86      // return commission rate
87      public double getCommissionRate()
88      {
89         return commissionRate;
90      } // end method getCommissionRate
91
92      // calculate earnings
93      public double earnings()
94      {
95         return commissionRate * grossSales;
96      } // end method earnings
97
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 5 of 6.)

```
98      // return String representation of CommissionEmployee object
99      @Override // indicates that this method overrides a superclass method
100     public String toString()
101     {
102        return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103           "commission employee", firstName, lastName,
104           "social security number", socialSecurityNumber,
105           "gross sales", grossSales,
106           "commission rate", commissionRate );
107     } // end method toString
108  } // end class CommissionEmployee
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 6 of 6.)

Comments (Class CommissionEmployee (Fig. 9.4))

- Class CommissionEmployee (Fig. 9.4) extends class Object (from package java.lang).
- CommissionEmployee inherits Object's methods.
- If you don't explicitly specify which class a new class extends, the class extends Object implicitly.
- Constructors are not inherited.
- The first task of a subclass constructor is to call its direct superclass's constructor explicitly or implicitly
- Ensures that the instance variables inherited from the superclass are initialized properly.
- If the code does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default or no-argument constructor.

- A class's default constructor calls the superclass's default or no-argument constructor.
- toString is one of the methods that every class inherits directly or indirectly from class Object.
- Returns a String representing an object.
- Called implicitly whenever an object must be converted to a String representation.
- Class Object's toString method returns a String that includes the name of the object's class.
- This is primarily a placeholder that can be overridden by a subclass to specify an appropriate String representation.
- To override a superclass method, a subclass must declare a method with the same signature as the superclass method
- @Override annotation
- Indicates that a method should override a superclass method with the same signature. If it does not, a compilation error occurs.

Testing CommissionEmployee Class

```
1  // Fig. 9.5: CommissionEmployeeTest.java
2  // CommissionEmployee class test program.
3
4  public class CommissionEmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // instantiate CommissionEmployee object
9        CommissionEmployee employee = new CommissionEmployee(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // get commission employee data
13       System.out.println(
14          "Employee information obtained by get methods: \n" );
15       System.out.printf( "%s %s\n", "First name is",
16          employee.getFirstName() );
17       System.out.printf( "%s %s\n", "Last name is",
18          employee.getLastName() );
19       System.out.printf( "%s %s\n", "Social security number is",
20          employee.getSocialSecurityNumber() );
21       System.out.printf( "%s %.2f\n", "Gross sales is",
22          employee.getGrossSales() );
23       System.out.printf( "%s %.2f\n", "Commission rate is",
24          employee.getCommissionRate() );
```

Fig. 9.5 | `CommissionEmployee` class test program. (Part 1 of 2.)

```
25
26          employee.setGrossSales( 500 ); // set gross sales
27          employee.setCommissionRate( .1 ); // set commission rate
28
29          System.out.printf( "\n%s:\n\n%s\n",
30             "Updated employee information obtained by toString", employee );
31       } // end main
32    } // end class CommissionEmployeeTest
```

Implicit `toString` call occurs here

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Fig. 9.5 | `CommissionEmployee` class test program. (Part 2 of 2.)

Creating and Using a BasePlus-CommissionEmployee Class

```
1  // Fig. 9.6: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class represents an employee who receives
3  // a base salary in addition to commission.
4
5  public class BasePlusCommissionEmployee
6  {
7     private String firstName;
8     private String lastName;
9     private String socialSecurityNumber;
10    private double grossSales; // gross weekly sales
11    private double commissionRate; // commission percentage
12    private double baseSalary; // base salary per week
13
14    // six-argument constructor
15    public BasePlusCommissionEmployee( String first, String last,
16       String ssn, double sales, double rate, double salary )
17    {
18       // implicit call to Object constructor occurs here
19       firstName = first;
20       lastName = last;
21       socialSecurityNumber = ssn;
22       setGrossSales( sales ); // validate and store gross sales
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 7.)

```
23          setCommissionRate( rate ); // validate and store commission rate
24          setBaseSalary( salary ); // validate and store base salary
25       } // end six-argument BasePlusCommissionEmployee constructor
26
27       // set first name
28       public void setFirstName( String first )
29       {
30          firstName = first; // should validate
31       } // end method setFirstName
32
33       // return first name
34       public String getFirstName()
35       {
36          return firstName;
37       } // end method getFirstName
38
39       // set last name
40       public void setLastName( String last )
41       {
42          lastName = last; // should validate
43       } // end method setLastName
44
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 7.)

```
45       // return last name
46       public String getLastName()
47       {
48          return lastName;
49       } // end method getLastName
50
51       // set social security number
52       public void setSocialSecurityNumber( String ssn )
53       {
54          socialSecurityNumber = ssn; // should validate
55       } // end method setSocialSecurityNumber
56
57       // return social security number
58       public String getSocialSecurityNumber()
59       {
60          return socialSecurityNumber;
61       } // end method getSocialSecurityNumber
62
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 7.)

```
63      // set gross sales amount
64      public void setGrossSales( double sales )
65      {
66         if ( sales >= 0.0 )
67            grossSales = sales;
68         else
69            throw new IllegalArgumentException(
70               "Gross sales must be >= 0.0" );
71      } // end method setGrossSales
72
73      // return gross sales amount
74      public double getGrossSales()
75      {
76         return grossSales;
77      } // end method getGrossSales
78
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 4 of 7.)

```
79      // set commission rate
80      public void setCommissionRate( double rate )
81      {
82         if ( rate > 0.0 && rate < 1.0 )
83            commissionRate = rate;
84         else
85            throw new IllegalArgumentException(
86               "Commission rate must be > 0.0 and < 1.0" );
87      } // end method setCommissionRate
88
89      // return commission rate
90      public double getCommissionRate()
91      {
92         return commissionRate;
93      } // end method getCommissionRate
94
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 5 of 7.)

```
95      // set base salary
96      public void setBaseSalary( double salary )
97      {
98         if ( salary >= 0.0 )
99            baseSalary = salary;
100        else
101           throw new IllegalArgumentException(
102              "Base salary must be >= 0.0" );
103     } // end method setBaseSalary
104
105     // return base salary
106     public double getBaseSalary()
107     {
108        return baseSalary;
109     } // end method getBaseSalary
110
111     // calculate earnings
112     public double earnings()
113     {
114        return baseSalary + ( commissionRate * grossSales );
115     } // end method earnings
116
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 6 of 7.)

```
117     // return String representation of BasePlusCommissionEmployee
118     @Override // indicates that this method overrides a superclass method
119     public String toString()
120     {
121        return String.format(
122           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
123           "base-salaried commission employee", firstName, lastName,
124           "social security number", socialSecurityNumber,
125           "gross sales", grossSales, "commission rate", commissionRate,
126           "base salary", baseSalary );
127     } // end method toString
128  } // end class BasePlusCommissionEmployee
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 7 of 7.)

Comments (BasePlusCommissionEmployee (Fig. 9.6))

- Class BasePlusCommissionEmployee (Fig. 9.6) contains a first name, last name, social security number, gross sales amount, commission rate and base salary.
- All but the base salary are in common with class CommissionEmployee.
- Class BasePlusCommissionEmployee's public services include a constructor, and methods earnings, toString and get and set for each instance variable
- Most of these are in common with class CommissionEmployee.

- Class BasePlusCommissionEmployee does not specify “extends Object”. It Implicitly extends Object class.
- BasePlusCommissionEmployee’s constructor invokes class Object’s default constructor implicitly.
- Much of BasePlusCommissionEmployee’s code is similar, or identical, to that of CommissionEmployee.
- private instance variables firstName and lastName and methods setFirstName, getFirstName, setLastName and getLastName are identical.
- Both classes also contain corresponding get and set methods.
- The constructors are almost identical
- BasePlusCommissionEmployee’s constructor also sets the base-Salary.
- The toString methods are nearly identical
- BasePlusCommissionEmployee’s toString also outputs instance variable baseSalary
- We literally copied CommissionEmployee’s code, pasted it into BasePlusCommissionEmployee, then modified the new class to include a base salary and methods that manipulate the base salary.
- This “copy-and-paste” approach is often error prone and time consuming.
- It spreads copies of the same code throughout a system, creating a code-maintenance nightmare.

Testing BasePlusCommissionEmployee

```
1  // Fig. 9.7: BasePlusCommissionEmployeeTest.java
2  // BasePlusCommissionEmployee test program.
3
4  public class BasePlusCommissionEmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // instantiate BasePlusCommissionEmployee object
9        BasePlusCommissionEmployee employee =
10          new BasePlusCommissionEmployee(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // get base-salaried commission employee data
14       System.out.println(
15          "Employee information obtained by get methods: \n" );
16       System.out.printf( "%s %s\n", "First name is",
17          employee.getFirstName() );
18       System.out.printf( "%s %s\n", "Last name is",
19          employee.getLastName() );
20       System.out.printf( "%s %s\n", "Social security number is",
21          employee.getSocialSecurityNumber() );
22       System.out.printf( "%s %.2f\n", "Gross sales is",
23          employee.getGrossSales() );
```

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 1 of 3.)

```
24         System.out.printf( "%s %.2f\n", "Commission rate is",
25            employee.getCommissionRate() );
26         System.out.printf( "%s %.2f\n", "Base salary is",
27            employee.getBaseSalary() );
28
29         employee.setBaseSalary( 1000 ); // set base salary
30
31         System.out.printf( "\n%s:\n\n%s\n",
32            "Updated employee information obtained by toString",
33             employee.toString() );
34      } // end main
35   } // end class BasePlusCommissionEmployeeTest
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 2 of 3.)

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 3 of 3.)

Case Study (CommissionEmployee–BasePlusCommissionEmployee using Inheritance Hierarchy)

Scenario-1 (private data members of supper class)

- Class BasePlusCommissionEmployee class extends class CommissionEmployee
- A BasePlusCommissionEmployee object is a CommissionEmployee
- Inheritance passes on class CommissionEmployee's capabilities.
- Class BasePlusCommissionEmployee also has instance variable baseSalary.
- Subclass BasePlusCommissionEmployee inherits CommissionEmployee's instance variables and methods
- Only the superclass's public and protected members are directly accessible in the subclass.
- Each subclass constructor must implicitly or explicitly call its superclass constructor to initialize the instance variables inherited from the superclass.
- Superclass constructor call syntax—keyword super, followed by a set of parentheses containing the superclass constructor arguments.

- Must be the first statement in the subclass constructor's body.
- If the subclass constructor did not invoke the superclass's constructor explicitly, Java would attempt to invoke the superclass's no-argument or default constructor.
- Class CommissionEmployee does not have such a constructor, so the compiler would issue an error.
- You can explicitly use super() to call the superclass's no-argument or default constructor, but this is rarely done.
- Compilation errors occur when the subclass attempts to access the superclass's private instance variables.

```
1  // Fig. 9.8: BasePlusCommissionEmployee.java
2  // private superclass members cannot be accessed in a subclass.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       // explicit call to superclass CommissionEmployee constructor
13       super( first, last, ssn, sales, rate );
14
15       setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 1 of 5.)

```
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       if ( salary >= 0.0 )
22          baseSalary = salary;
23       else
24          throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26    } // end method setBaseSalary
27
28    // return base salary
29    public double getBaseSalary()
30    {
31       return baseSalary;
32    } // end method getBaseSalary
33
34    // calculate earnings
35    @Override // indicates that this method overrides a superclass method
36    public double earnings()
37    {
38       // not allowed: commissionRate and grossSales private in superclass
39       return baseSalary + ( commissionRate * grossSales );
40    } // end method earnings
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 2 of 5.)

```
41
42     // return String representation of BasePlusCommissionEmployee
43     @Override // indicates that this method overrides a superclass method
44     public String toString()
45     {
46        // not allowed: attempts to access private superclass members
47        return String.format(
48           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
49           "base-salaried commission employee", firstName, lastName,
50           "social security number", socialSecurityNumber,
51           "gross sales", grossSales, "commission rate", commissionRate,
52           "base salary", baseSalary );
53     } // end method toString
54  } // end class BasePlusCommissionEmployee
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 3 of 5.)

Scenario-2 (protected data members of supper class)

- To enable a subclass to directly access superclass instance variables, we can declare those members as protected in the superclass.
- New CommissionEmployee class modified only lines 6–10 of Fig. 9.4 as follows:

```
protected String firstName;
protected String lastName;
protected String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

- With protected instance variables, the subclass gets access to the instance variables, but classes that are not subclasses and classes that are not in the same package cannot access these variables directly.
- Class BasePlusCommissionEmployee (Fig. 9.9) extends the new version of class CommissionEmployee with protected instance variables.
- These variables are now protected members of BasePlusCommissionEmployee.
- If another class extends this version of class BasePlusCommissionEmployee, the new subclass also can access the protected members.
- The source code in Fig. 9.9 (51 lines) is considerably shorter than that in Fig. 9.6 (128 lines)
- Most of the functionality is now inherited from CommissionEmployee
- There is now only one copy of the functionality.
- Code is easier to maintain, modify and debug—the code related to a commission employee exists only in class CommissionEmployee.
- Inheriting protected instance variables slightly increases performance, because we can directly access the variables in the subclass without incurring the overhead of a set or get method call.
- In most cases, it's better to use private instance variables to encourage proper software engineering, and leave code optimization issues to the compiler. Code will be easier to maintain, modify and debug.

- Using protected instance variables creates several potential problems.
- The subclass object can set an inherited variable's value directly without using a set method.
- A subclass object can assign an invalid value to the variable
- Subclass methods are more likely to be written so that they depend on the superclass's data implementation.
- Subclasses should depend only on the superclass services and not on the superclass data implementation.
- With protected instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes.
- Such software is said to be fragile or brittle, because a small change in the superclass can "break" subclass implementation.
- You should be able to change the superclass implementation while still providing the same services to the subclasses.
- If the superclass services change, we must re-implement our subclasses.
- A class's protected members are visible to all classes in the same package as the class containing the protected members—this is not always desirable.

```
1  // Fig. 9.9: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee inherits protected instance
3  // variables from CommissionEmployee.
4
5  public class BasePlusCommissionEmployee extends CommissionEmployee
6  {
7     private double baseSalary; // base salary per week
8
9     // six-argument constructor
10    public BasePlusCommissionEmployee( String first, String last,
11       String ssn, double sales, double rate, double salary )
12    {
13       super( first, last, ssn, sales, rate );
14       setBaseSalary( salary ); // validate and store base salary
15    } // end six-argument BasePlusCommissionEmployee constructor
16
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 1 of 3.)

```
17      // set base salary
18      public void setBaseSalary( double salary )
19      {
20         if ( salary >= 0.0 )
21            baseSalary = salary;
22         else
23            throw new IllegalArgumentException(
24               "Base salary must be >= 0.0" );
25      } // end method setBaseSalary
26
27      // return base salary
28      public double getBaseSalary()
29      {
30         return baseSalary;
31      } // end method getBaseSalary
32
33      // calculate earnings
34      @Override // indicates that this method overrides a superclass method
35      public double earnings()
36      {
37         return baseSalary + ( commissionRate * grossSales );
38      } // end method earnings
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 2 of 3.)

```
39
40      // return String representation of BasePlusCommissionEmployee
41      @Override // indicates that this method overrides a superclass method
42      public String toString()
43      {
44         return String.format(
45            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
46            "base-salaried commission employee", firstName, lastName,
47            "social security number", socialSecurityNumber,
48            "gross sales", grossSales, "commission rate", commissionRate,
49            "base salary", baseSalary );
50      } // end method toString
51   } // end class BasePlusCommissionEmployee
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 3 of 3.)

Scenario-3 (CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables) Re-engineering of supper class

- Hierarchy reengineered using good software engineering practices.
- Class CommissionEmployee declares instance variables firstName, lastName, socialSecurityNumber, grossSales and commissionRate as private and provides public methods for manipulating these values.
- CommissionEmployee methods earnings and toString use the class's get methods to obtain the values of its instance variables.

- If we decide to change the internal representation of the data (e.g., variable names) only the bodies of the get and set methods that directly manipulate the instance variables will need to change.
- These changes occur solely within the superclass-—no changes to the subclass are needed.
- Localizing the effects of changes like this is a good software engineering practice.
- Subclass BasePlusCommissionEmployee inherits Commission-Employee's non-private methods and can access the private superclass members via those methods.

```
1  // Fig. 9.10: CommissionEmployee.java
2  // CommissionEmployee class uses methods to manipulate its
3  // private instance variables.
4  public class CommissionEmployee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs here
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate and store gross sales
21       setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 1 of 6.)

```
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first; // should validate
28    } // end method setFirstName
29
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       la5stName = last; // should validate
40    } // end method setLastName
41
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 2 of 6.)

```
42      // return last name
43      public String getLastName()
44      {
45         return lastName;
46      } // end method getLastName
47
48      // set social security number
49      public void setSocialSecurityNumber( String ssn )
50      {
51         socialSecurityNumber = ssn; // should validate
52      } // end method setSocialSecurityNumber
53
54      // return social security number
55      public String getSocialSecurityNumber()
56      {
57         return socialSecurityNumber;
58      } // end method getSocialSecurityNumber
59
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 3 of 6.)

```
60      // set gross sales amount
61      public void setGrossSales( double sales )
62      {
63         if ( sales >= 0.0 )
64            grossSales = sales;
65         else
66            throw new IllegalArgumentException(
67               "Gross sales must be >= 0.0" );
68      } // end method setGrossSales
69
70      // return gross sales amount
71      public double getGrossSales()
72      {
73         return grossSales;
74      } // end method getGrossSales
75
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 4 of 6.)

```
76      // set commission rate
77      public void setCommissionRate( double rate )
78      {
79         if ( rate > 0.0 && rate < 1.0 )
80            commissionRate = rate;
81         else
82            throw new IllegalArgumentException(
83               "Commission rate must be > 0.0 and < 1.0" );
84      } // end method setCommissionRate
85
86      // return commission rate
87      public double getCommissionRate()
88      {
89         return commissionRate;
90      } // end method getCommissionRate
91
92      // calculate earnings
93      public double earnings()
94      {
95         return getCommissionRate() * getGrossSales();
96      } // end method earnings
97
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 5 of 6.)

```
98      // return String representation of CommissionEmployee object
99      @Override // indicates that this method overrides a superclass method
100     public String toString()
101     {
102        return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103           "commission employee", getFirstName(), getLastName(),
104           "social security number", getSocialSecurityNumber(),
105           "gross sales", getGrossSales(),
106           "commission rate", getCommissionRate() );
107     } // end method toString
108  } // end class CommissionEmployee
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 6 of 6.)

```
1  // Fig. 9.11: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class inherits from CommissionEmployee
3  // and accesses the superclass's private data via inherited
4  // public methods.
5
6  public class BasePlusCommissionEmployee extends CommissionEmployee
7  {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee( String first, String last,
12       String ssn, double sales, double rate, double salary )
13    {
14       super( first, last, ssn, sales, rate );
15       setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 1 of 3.)

```
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       if ( salary >= 0.0 )
22          baseSalary = salary;
23       else
24          throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26    } // end method setBaseSalary
27
28    // return base salary
29    public double getBaseSalary()
30    {
31       return baseSalary;
32    } // end method getBaseSalary
33
34    // calculate earnings
35    @Override // indicates that this method overrides a superclass method
36    public double earnings()
37    {
38       return getBaseSalary() + super.earnings();
39    } // end method earnings
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 2 of 3.)

```
40
41      // return String representation of BasePlusCommissionEmployee
42      @Override // indicates that this method overrides a superclass method
43      public String toString()
44      {
45         return String.format( "%s %s\n%s: %.2f", "base-salaried",
46            super.toString(), "base salary", getBaseSalary() );
47      } // end method toString
48   } // end class BasePlusCommissionEmployee
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 3 of 3.)

- Class BasePlusCommissionEmployee (Fig. 9.11) has several changes that distinguish it from Fig. 9.9.
- Methods earnings and toString each invoke their superclass versions and do not access instance variables directly.
- Method earnings overrides class the superclass's earnings method.
- The new version calls CommissionEmployee's earnings method with super.earnings().
- Obtains the earnings based on commission alone
- Placing the keyword super and a dot (.) separator before the superclass method name invokes the superclass version of an overridden method.
- Good software engineering practice
- If a method performs all or some of the actions needed by another method, call that method rather than duplicate its code.
- BasePlusCommissionEmployee's toString method overrides class CommissionEmployee's toString method.
- The new version creates part of the String representation by calling CommissionEmployee's toString method with the expression super.toString().

Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
- The subclass constructor, before performing its own tasks, invokes its direct superclass's constructor
- If the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up the hierarchy, and so on.
- The last constructor called in the chain is always class Object's constructor.
- Original subclass constructor's body finishes executing last.
- Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits.

Object Class

- All classes in Java inherit directly or indirectly from Object, so its 11 methods are inherited by all other classes.
- Figure 9.12 summarizes Object's methods.
- Every array has an overridden clone method that copies the array.
- If the array stores references to objects, the objects are not copied—a shallow copy is performed.

Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation performs a so-called **shallow copy**—instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a **deep copy** that creates a new object for each reference-type instance variable. Implementing `clone` correctly is difficult. For this reason, its use is discouraged. Many industry experts suggest that object serialization should be used instead. We discuss object serialization in Chapter 17, Files, Streams and Object Serialization.

Fig. 9.12 | `Object` methods. (Part 1 of 3.)

Method	Description
`equals`	This method compares two objects for equality and returns `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the *contents* of the two objects. For the requirements of implementing this method, refer to the method's documentation at `download.oracle.com/javase/6/docs/api/java/lang/Object.html# equals(java.lang.Object)`. The default `equals` implementation uses operator `==` to determine whether two references *refer to the same object* in memory. Section 16.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with `==` and with `equals`.
`finalize`	This `protected` method (introduced in Section 8.10) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall that it's unclear whether, or when, method `finalize` will be called. For this reason, most programmers should avoid method `finalize`.

Fig. 9.12 | `Object` methods. (Part 2 of 3.)

Method	Description
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Sections 10.5, 14.5 and 24.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`).
`hashCode`	Hashcodes are `int` values that are useful for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (discussed in Section 20.11). This method is also called as part of class `Object`'s default `toString` method implementation.
`wait`, `notify`, `notifyAll`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 26.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Fig. 9.12 | `Object` methods. (Part 3 of 3.)

2. Polymorphism

- Enables you to “program in the general” rather than “program in the specific.”
- Polymorphism enables you to write programs that process objects that share the same superclass as if they’re all objects of the superclass; this can simplify programming.
- With polymorphism, we can design and implement systems that are easily extensible
- New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
- The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.
-

Interfaces

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

Polymorphism Examples

Example 1:

- 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.
- 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.

Example 2:

- 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.

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.
- 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.

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.
- 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.
- 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.
- 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.
- 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.
- You make a class abstract by declaring it with keyword abstract.
- An abstract class normally contains one or more abstract methods.
- An abstract method is one with keyword abstract in its declara-tion, 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.
- Cannot instantiate objects of abstract superclasses, but you can use abstract superclasses to declare variables
- These can hold references to objects of any concrete class derived from those abstract superclasses.
- Programs typically use such variables to manipulate subclass objects polymorphically.
- Can use abstract superclass names to invoke static methods declared in those abstract superclasses.
- 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.
- 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.

Case Study: Payroll System Using Polymorphism

- 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 salaried-commission employees by adding 10% to their base salaries. The company wants to write a Java application that performs its payroll calculations polymorphically.

- 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.
- 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.

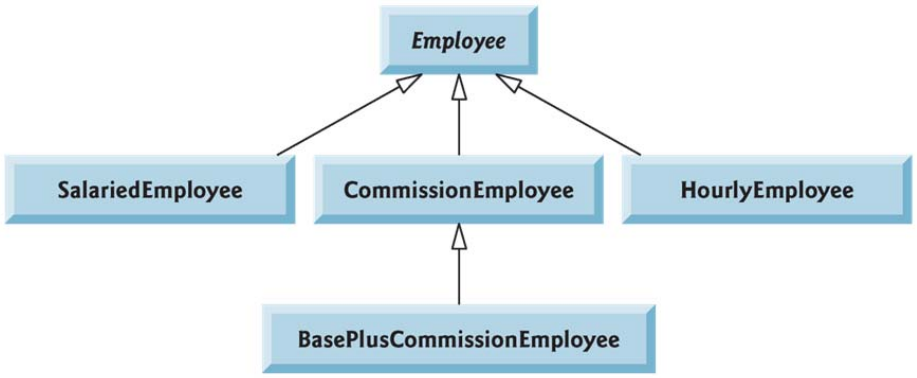

Fig. 10.2 | `Employee` hierarchy UML class diagram.

	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.

- 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.
- 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.

Abstract Superclass `Employee`

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first; // should validate
22    } // end method setFirstName
23
```

Fig. 10.4 | `Employee` abstract superclass. (Part 1 of 3.)

```
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last; // should validate
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
```

Fig. 10.4 | `Employee` abstract superclass. (Part 2 of 3.)

```
48     // return social security number
49     public String getSocialSecurityNumber()
50     {
51        return socialSecurityNumber;
52     } // end method getSocialSecurityNumber
53
54     // return String representation of Employee object
55     @Override
56     public String toString()
57     {
58        return String.format( "%s %s\nsocial security number: %s",
59           getFirstName(), getLastName(), getSocialSecurityNumber() );
60     } // end method toString
61
62     // abstract method overridden by concrete subclasses
63     public abstract double earnings(); // no implementation here
64  } // end abstract class Employee
```

This method must be overridden in subclasses to make them concrete

Fig. 10.4 | `Employee` abstract superclass. (Part 3 of 3.)

Concrete Subclass `SalariedEmployee`

```
1  // Fig. 10.5: SalariedEmployee.java
2  // SalariedEmployee concrete class extends abstract class Employee.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // four-argument constructor
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // pass to Employee constructor
13       setWeeklySalary( salary ); // validate and store salary
14    } // end four-argument SalariedEmployee constructor
15
```

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

```
16      // set salary
17      public void setWeeklySalary( double salary )
18      {
19         if ( salary >= 0.0 )
20            baseSalary = salary;
21         else
22            throw new IllegalArgumentException(
23               "Weekly salary must be >= 0.0" );
24      } // end method setWeeklySalary
25
26      // return salary
27      public double getWeeklySalary()
28      {
29         return weeklySalary;
30      } // end method getWeeklySalary
31
```

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

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

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

Concrete Subclass HourlyEmployee

```
1  // Fig. 10.6: HourlyEmployee.java
2  // HourlyEmployee class extends Employee.
3
4  public class HourlyEmployee extends Employee
5  {
6     private double wage; // wage per hour
7     private double hours; // hours worked for week
8
9     // five-argument constructor
10    public HourlyEmployee( String first, String last, String ssn,
11       double hourlyWage, double hoursWorked )
12    {
13       super( first, last, ssn );
14       setWage( hourlyWage ); // validate hourly wage
15       setHours( hoursWorked ); // validate hours worked
16    } // end five-argument HourlyEmployee constructor
17
```

Fig. 10.6 | HourlyEmployee class extends Employee. (Part 1 of 4.)

```
18    // set wage
19    public void setWage( double hourlyWage )
20    {
21       if ( hourlyWage >= 0.0 )
22          wage = hourlyWage;
23       else
24          throw new IllegalArgumentException(
25             "Hourly wage must be >= 0.0" );
26    } // end method setWage
27
28    // return wage
29    public double getWage()
30    {
31       return wage;
32    } // end method getWage
33
```

Fig. 10.6 | HourlyEmployee class extends Employee. (Part 2 of 4.)

```
34      // set hours worked
35      public void setHours( double hoursWorked )
36      {
37         if ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) )
38            hours = hoursWorked;
39         else
40            throw new IllegalArgumentException(
41               "Hours worked must be >= 0.0 and <= 168.0" );
42      } // end method setHours
43
44      // return hours worked
45      public double getHours()
46      {
47         return hours;
48      } // end method getHours
49
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 3 of 4.)

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

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

Concrete Subclass CommissionEmployee

```
1  // Fig. 10.7: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3
4  public class CommissionEmployee extends Employee
5  {
6     private double grossSales; // gross weekly sales
7     private double commissionRate; // commission percentage
8
9     // five-argument constructor
10    public CommissionEmployee( String first, String last, String ssn,
11       double sales, double rate )
12    {
13       super( first, last, ssn );
14       setGrossSales( sales );
15       setCommissionRate( rate );
16    } // end five-argument CommissionEmployee constructor
17
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 1 of 4.)

```
18    // set commission rate
19    public void setCommissionRate( double rate )
20    {
21       if ( rate > 0.0 && rate < 1.0 )
22          commissionRate = rate;
23       else
24          throw new IllegalArgumentException(
25             "Commission rate must be > 0.0 and < 1.0" );
26    } // end method setCommissionRate
27
28    // return commission rate
29    public double getCommissionRate()
30    {
31       return commissionRate;
32    } // end method getCommissionRate
33
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 2 of 4.)

```
34      // set gross sales amount
35      public void setGrossSales( double sales )
36      {
37         if ( sales >= 0.0 )
38            grossSales = sales;
39         else
40            throw new IllegalArgumentException(
41               "Gross sales must be >= 0.0" );
42      } // end method setGrossSales
43
44      // return gross sales amount
45      public double getGrossSales()
46      {
47         return grossSales;
48      } // end method getGrossSales
49
50      // calculate earnings; override abstract method earnings in Employee
51      @Override
52      public double earnings()
53      {
54         return getCommissionRate() * getGrossSales();
55      } // end method earnings
56
```

Fig. 10.7 | `CommissionEmployee` class extends `Employee`. (Part 3 of 4.)

```
57      // return String representation of CommissionEmployee object
58      @Override
59      public String toString()
60      {
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      } // end method toString
66   } // end class CommissionEmployee
```

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

Concrete Subclass BasePlusCommissionEmployee

```
1  // Fig. 10.8: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class extends CommissionEmployee.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       super( first, last, ssn, sales, rate );
13       setBaseSalary( salary ); // validate and store base salary
14    } // end six-argument BasePlusCommissionEmployee constructor
15
```

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

```
16    // set base salary
17    public void setBaseSalary( double salary )
18    {
19       if ( salary >= 0.0 )
20          baseSalary = salary;
21       else
22          throw new IllegalArgumentException(
23             "Base salary must be >= 0.0" );
24    } // end method setBaseSalary
25
26    // return base salary
27    public double getBaseSalary()
28    {
29       return baseSalary;
30    } // end method getBaseSalary
31
```

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

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

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

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   {
6      public static void main( String[] args )
7      {
8         // create subclass objects
9         SalariedEmployee salariedEmployee =
10           new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11        HourlyEmployee hourlyEmployee =
12           new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13        CommissionEmployee commissionEmployee =
14           new CommissionEmployee(
15           "Sue", "Jones", "333-33-3333", 10000, .06 );
16        BasePlusCommissionEmployee basePlusCommissionEmployee =
17           new BasePlusCommissionEmployee(
18           "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 1 of 7.)

```
20          System.out.println( "Employees processed individually:\n" );
21
22          System.out.printf( "%s\n%s: $%,.2f\n\n",
23             salariedEmployee, "earned", salariedEmployee.earnings() );
24          System.out.printf( "%s\n%s: $%,.2f\n\n",
25             hourlyEmployee, "earned", hourlyEmployee.earnings() );
26          System.out.printf( "%s\n%s: $%,.2f\n\n",
27             commissionEmployee, "earned", commissionEmployee.earnings() );
28          System.out.printf( "%s\n%s: $%,.2f\n\n",
29             basePlusCommissionEmployee,
30             "earned", basePlusCommissionEmployee.earnings() );
31
32          // create four-element Employee array
33          Employee[] employees = new Employee[ 4 ];
34
35          // initialize array with Employees
36          employees[ 0 ] = salariedEmployee;
37          employees[ 1 ] = hourlyEmployee;
38          employees[ 2 ] = commissionEmployee;
39          employees[ 3 ] = basePlusCommissionEmployee;
40
41          System.out.println( "Employees processed polymorphically:\n" );
42
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 2 of 7.)

```
43          // generically process each element in array employees
44          for ( Employee currentEmployee : employees )
45          {
46             System.out.println( currentEmployee ); // invokes toString
47
48             // determine whether element is a BasePlusCommissionEmployee
49             if ( currentEmployee instanceof BasePlusCommissionEmployee )
50             {
51                // downcast Employee reference to
52                // BasePlusCommissionEmployee reference
53                BasePlusCommissionEmployee employee =
54                   ( BasePlusCommissionEmployee ) currentEmployee;
55
56                employee.setBaseSalary( 1.10 * employee.getBaseSalary() );
57
58                System.out.printf(
59                   "new base salary with 10%% increase is: $%,.2f\n",
60                   employee.getBaseSalary() );
61             } // end if
62
63             System.out.printf(
64                "earned $%,.2f\n\n", currentEmployee.earnings() );
65          } // end for
66
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 3 of 7.)

```
67          // get type name of each object in employees array
68          for ( int j = 0; j < employees.length; j++ )
69             System.out.printf( "Employee %d is a %s\n", j,
70                employees[ j ].getClass().getName() );
71       } // end main
72    } // end class PayrollSystemTest
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 4 of 7.)

```
Employees processed individually:

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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 5 of 7.)

```
Employees processed polymorphically:

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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 6 of 7.)

```
Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

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

- Finally, the program polymorphically determines and outputs the type of each object in the Employee array.
- 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.

Summary of the Allowed Assignments Between Superclass and Subclass Variables

- There are four ways to assign superclass and subclass references to variables of superclass and subclass types.
- Assigning a superclass reference to a superclass variable is straightforward.
- Assigning a subclass reference to a subclass variable is 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.
- Attempting to assign a superclass reference to a subclassvariable is a compilation error.
- To avoid this error, the superclass reference must be cast to a subclass type explicitly.
- At execution time, if the object to which the reference refers is not a subclass object, an exception will occur.
- Use the instanceof operator to ensure that such a cast is performed only if the object is a subclass object.

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

- Since class String cannot be extended, programs that use Strings 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.

3. Interfaces

- Interfaces offer a capability requiring that unrelated classes implement a set of common methods.
- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
- Example: The controls on a radio serve as an interface between radio users and a radio's internal components.
- Can perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM)
- Different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands).
- The interface specifies what operations a radio must permit users to perform but does not specify how the operations are performed.
- A Java interface describes a set of methods that can be called on an object.
- 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.
- 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."
- An interface is often used when disparate (i.e., unrelated) classes need to share common methods and constants.
- Allows objects of unrelated classes to be processed polymorphically by responding to the same method calls.
- You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality.
- An interface is often used in place of an abstract class when there is no default implementation to inherit—that is, no fields and no default method implementations.
- 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 file-name extension.
-

3. Case Study - Creating and Using Interfaces

- 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.
- 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.
- 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.

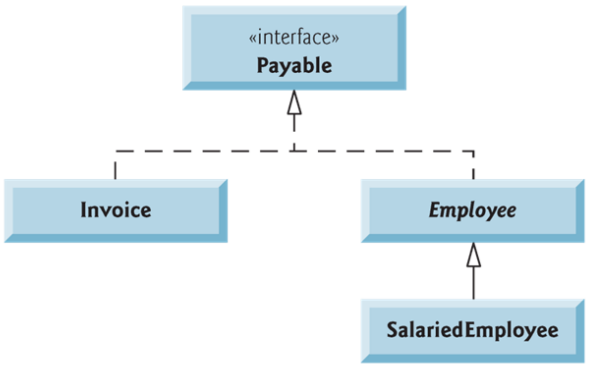

Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

Interface `Payable`

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

Fig. 10.11 | `Payable` interface declaration.

- 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 fields that are implicitly final and static.

Class Invoice to Implement Interface `Payable`

```
1  // Fig. 10.12: Invoice.java
2  // Invoice class that implements Payable.
3
4  public class Invoice implements Payable
5  {
6     private String partNumber;
7     private String partDescription;
8     private int quantity;
9     private double pricePerItem;
10
11    // four-argument constructor
12    public Invoice( String part, String description, int count,
13       double price )
14    {
15       partNumber = part;
16       partDescription = description;
17       setQuantity( count ); // validate and store quantity
18       setPricePerItem( price ); // validate and store price per item
19    } // end four-argument Invoice constructor
20
```

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

```
21    // set part number
22    public void setPartNumber( String part )
23    {
24       partNumber = part; // should validate
25    } // end method setPartNumber
26
27    // get part number
28    public String getPartNumber()
29    {
30       return partNumber;
31    } // end method getPartNumber
32
33    // set description
34    public void setPartDescription( String description )
35    {
36       partDescription = description; // should validate
37    } // end method setPartDescription
38
39    // get description
40    public String getPartDescription()
41    {
42       return partDescription;
43    } // end method getPartDescription
44
```

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

```
45      // set quantity
46      public void setQuantity( int count )
47      {
48         if ( count >= 0 )
49            quantity = count;
50         else
51            throw new IllegalArgumentException( "Quantity must be >= 0" );
52      } // end method setQuantity
53
54      // get quantity
55      public int getQuantity()
56      {
57         return quantity;
58      } // end method getQuantity
59
60      // set price per item
61      public void setPricePerItem( double price )
62      {
63         if ( price >= 0.0 )
64            pricePerItem = price;
65         else
66            throw new IllegalArgumentException(
67               "Price per item must be >= 0" );
68      } // end method setPricePerItem
```

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

```
69
70      // get price per item
71      public double getPricePerItem()
72      {
73         return pricePerItem;
74      } // end method getPricePerItem
75
76      // return String representation of Invoice object
77      @Override
78      public String toString()
79      {
80         return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
81            "invoice", "part number", getPartNumber(), getPartDescription(),
82            "quantity", getQuantity(), "price per item", getPricePerItem() );
83      } // end method toString
84
85      // method required to carry out contract with interface Payable
86      @Override
87      public double getPaymentAmount()
88      {
89         return getQuantity() * getPricePerItem(); // calculate total cost
90      } // end method getPaymentAmount
91   } // end class Invoice
```

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

Class `Employee` to Implement Interface `Payable`

```
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass that implements Payable.
3
4  public abstract class Employee implements Payable
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first; // should validate
22    } // end method setFirstName
23
```

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

```
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last; // should validate
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
```

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

```
48      // return social security number
49      public String getSocialSecurityNumber()
50      {
51         return socialSecurityNumber;
52      } // end method getSocialSecurityNumber
53
54      // return String representation of Employee object
55      @Override
56      public String toString()
57      {
58         return String.format( "%s %s\nsocial security number: %s",
59            getFirstName(), getLastName(), getSocialSecurityNumber() );
60      } // end method toString
61
62      // Note: We do not implement Payable method getPaymentAmount here so
63      // this class must be declared abstract to avoid a compilation error.
64   } // end abstract class Employee
```

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

Class SalariedEmployee for Use in the Payable Hierarchy

```
1   // Fig. 10.14: SalariedEmployee.java
2   // SalariedEmployee class extends Employee, which implements Payable.
3
4   public class SalariedEmployee extends Employee
5   {
6      private double weeklySalary;
7
8      // four-argument constructor
9      public SalariedEmployee( String first, String last, String ssn,
10        double salary )
11     {
12        super( first, last, ssn ); // pass to Employee constructor
13        setWeeklySalary( salary ); // validate and store salary
14     } // end four-argument SalariedEmployee constructor
15
```

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

```
16      // set salary
17      public void setWeeklySalary( double salary )
18      {
19         if ( salary >= 0.0 )
20            baseSalary = salary;
21         else
22            throw new IllegalArgumentException(
23               "Weekly salary must be >= 0.0" );
24      } // end method setWeeklySalary
25
26      // return salary
27      public double getWeeklySalary()
28      {
29         return weeklySalary;
30      } // end method getWeeklySalary
31
```

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

```
32      // calculate earnings; implement interface Payable method that was
33      // abstract in superclass Employee
34      @Override
35      public double getPaymentAmount()
36      {
37         return getWeeklySalary();
38      } // end method getPaymentAmount
39
40      // return String representation of SalariedEmployee object
41      @Override
42      public String toString()
43      {
44         return String.format( "salaried employee: %s\n%s: $%,.2f",
45            super.toString(), "weekly salary", getWeeklySalary() );
46      } // end method toString
47   } // end class SalariedEmployee
```

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

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

Class `PayableInterfaceTest`

-

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

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

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

```
Invoices and Employees processed polymorphically:

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

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

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

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

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

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

Common Interfaces of the Java API

- The Java API's interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program.
- Figure 10.16 presents a brief overview of a few of the more popular interfaces of the Java API .

Interface	Description
`Comparable`	Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators *cannot* be used to compare objects. Interface `Comparable` is used to allow objects of a class that `implements` the interface to be compared to one another. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 20, Generic Collections, and Chapter 21, Generic Classes and Methods.
`Serializable`	An interface used to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 17, Files, Streams and Object Serialization, and Chapter 27, Networking.
`Runnable`	Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Chapter 26, Multithreading). The interface contains one method, `run`, which describes the behavior of an object when executed.

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

Interface	Description
GUI event-listener interfaces	You work with graphical user interfaces (GUIs) every day. In your web browser, you might type the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler. In Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2, you'll learn how to build GUIs and event handlers that respond to user interactions. Event handlers are declared in classes that implement an appropriate event-listener interface. Each event-listener interface specifies one or more methods that must be implemented to respond to user interactions.
`SwingConstants`	Contains a set of constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 14 and 25.

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