# 10.5 Case Study: Payroll System Using Polymorphism

This section reexamines the `CommissionEmployee-BasePlusCommissionEmployee` hierarchy that we explored throughout Section 9.4. Now we use an abstract method and polymorphism to perform payroll calculations based on an enhanced employee inheritance hierarchy that meets the following requirements:

A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base-salaried commission employees by adding 10% to their base salaries. The company wants you to write an application that performs its payroll calculations polymorphically.

We use `abstract` class `Employee` to represent the general

concept of an employee. The classes that extend `Employee` are `SalariedEmployee`, `CommissionEmployee` and `Hourly-Employee.` Class `BasePlusCommissionEmployee`—which extends `CommissionEmployee`—represents the last employee type. The UML class diagram in Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application. Abstract class name `Employee` is *italicized*—a convention of the UML.
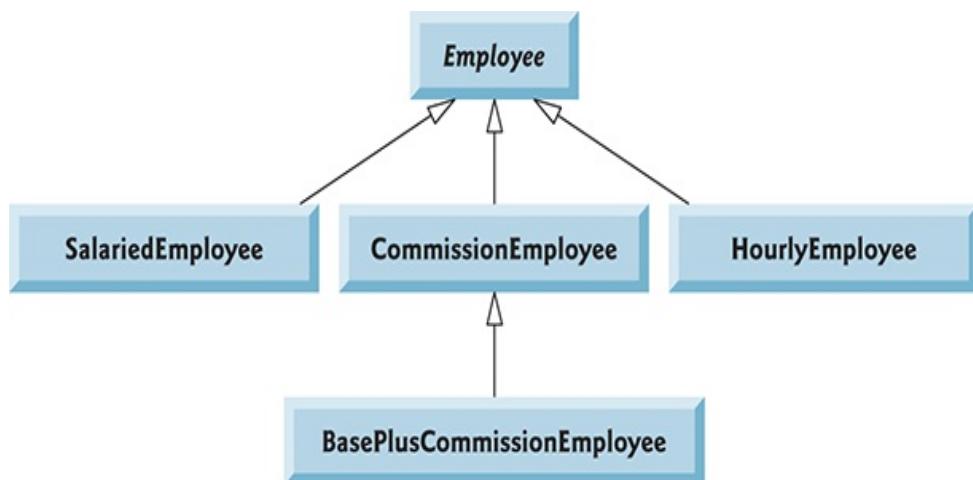
Fig. 10.2

`Employee` hierarchy UML class diagram.

Abstract superclass `Employee` declares the "interface" to the hierarchy—that is, the set of methods that a program can invoke on all `Employee` objects. We use the term "interface" here in a *general* sense to refer to the various ways programs can communicate with objects of *any* `Employee` subclass. Be

careful not to confuse the general notion of an "interface" with the formal notion of a Java interface, the subject of Section 10.9. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so `private` instance variables `firstName`, `lastName` and `socialSecurityNumber` appear in abstract superclass `Employee`.

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. We do not list superclass `Employee`'s *get* methods because they're *not* overridden in any of the subclasses—each of these methods is inherited and used "as is" by each subclass.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: S
Salaried-Employee	weeklySalary	salaried employee: *firstName l* social security number: S weekly salary: *weeklySala*
Hourly-Employee	if (hours <= 40) { wage * hours } else if (hours > 40) { 40 * wage + (hours - 40) *	hourly employee: *firstName la* social security number: S hourly wage: *wage*; hours worke

	`wage * 1.5` `}`	
	◀ ▶	

| Commission-
Employee | `commissionRate *`
`grossSales` | `commission employee: firstName`
`social security number: S`
`gross sales: grossSales`
`commission rate: commission`
◀ ▶ |
| BasePlus-
Commission-
Employee | `(commissionRate *`
`grossSales) +`
`baseSalary` | `base salaried commission emp`
`firstName lastName`
`social security number: S`
`gross sales: grossSales`
`commission rate: commissionl`
`base salary: baseSalary`
◀ ▶ |

Fig. 10.3

Polymorphic interface for the `Employee` hierarchy classes.

The following sections implement the `Employee` class hierarchy of Fig. 10.2. The first section implements *abstract superclass* `Employee`. The next four sections each implement one of the *concrete* classes. The last section implements a test program that builds objects of all these classes and processes those objects polymorphically.

10.5.1 Abstract Superclass Employee

Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString,` in addition to the *get* methods that return the values of `Employee`'s instance variables. An `earnings` method certainly applies *generically* to all employees. But each earnings calculation depends on the employee's particular class. So we declare `earnings` as `abstract` in superclass `Employee` because a *specific* default implementation does not make sense for that method—there isn't enough information to determine what amount `earnings` should return.

Each subclass overrides `earnings` with an appropriate implementation. To calculate an employee's earnings, the program assigns to a superclass `Employee` variable a reference to the employee's object, then invokes the `earnings` method on that variable. We maintain an array of `Employee` variables, each holding a reference to an `Employee` object. You *cannot* use class `Employee` directly to create `Employee` *objects*, because `Employee` is an *abstract* class. Due to inheritance, however, all objects of all `Employee` subclasses may be thought of as `Employee` objects. The program will iterate through the array and call method `earnings` for each `Employee` object. Java processes these method calls *polymorphically*. Declaring `earnings` as an `abstract` method in `Employee` enables the calls to `earnings` through `Employee` variables to

compile and forces every direct *concrete* subclass of `Employee` to *override* `earnings`.

Method `toString` in class `Employee` returns a `String` containing the first name, last name and social security number of the employee. As we'll see, each subclass of `Employee` *overrides* method `toString` to create a `String` representation of an object of that class that contains the employee's type (e.g., `"salaried employee:"`) followed by the rest of the employee's information.

Let's consider class `Employee`'s declaration (Fig. 10.4). The class includes a constructor that receives the first name, last name and social security number (lines 10–15); *get* methods that return the first name, last name and social security number (lines 18, 21 and 24, respectively); method `toString` (lines 27–31), which returns the `String` representation of an `Employee`; and `abstract` method `earnings` (line 34), which will be implemented by each of the *concrete* subclasses. The `Employee` constructor does *not* validate its parameters in this example; normally, such validation should be provided.

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8
9     // constructor
10    public Employee(String firstName, String lastN
```

```
11          String socialSecurityNumber) {
12            this.firstName = firstName;
13             this.lastName = lastName;
14         this.socialSecurityNumber = socialSecurityN
15         }
16
17      // return first name
18     public String getFirstName() {return firstName
19
20      // return last name
21     public String getLastName() {return lastName;}
22
23      // return social security number
24     public String getSocialSecurityNumber() {retur
25
26     // return String representation of Employee ob
27        @Override
28      public String toString() {
29        return String.format("%s %s%nsocial securit
30           getFirstName(), getLastName(), getSocial
31        }
32
33     // abstract method must be overridden by concr
34     public abstract double earnings(); // no imple
35   }
```
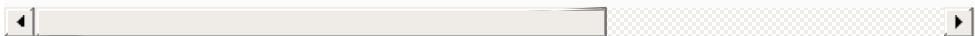
Fig. 10.4

`Employee` abstract superclass.

Why did we decide to declare `earnings` as an `abstract`
method? It simply does not make sense to provide a *specific*
implementation of this method in class `Employee`. We
cannot calculate the earnings for a *general* `Employee`—we

first must know the *specific* type of `Employee` to determine the appropriate earnings calculation. By declaring this method `abstract`, we indicate 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`.

10.5.2 Concrete Subclass SalariedEmployee

Class `SalariedEmployee` (Fig. 10.5) extends class `Employee` (line 4) and overrides *abstract* method `earnings` (lines 34–35), which makes `SalariedEmployee` a *concrete* class. The class includes a constructor (lines 8–18) that receives a first name, a last name, a social security number and a weekly salary; a *set* method to assign a new *nonnegative* value to instance variable `weeklySalary` (lines 21–28); a *get* method to return `weeklySalary`'s value (line 31); a method `earnings` (lines 34–35) to calculate a `SalariedEmployee`'s earnings; and a method `toString` (lines 38–42), which returns a `String` including `"salaried employee: "` followed by employee-specific information produced by superclass `Employee`'s `toString` method and `SalariedEmployee`'s `getWeeklySalary` method. Class `SalariedEmployee`'s constructor passes the first name, last name and social security number to the `Employee`

constructor (line 10) to initialize the `private` instance variables of the superclass. Once again, we've duplicated the `weeklySalary` validation code in the constructor and the `setWeeklySalary` method. Recall that more complex validation could be placed in a `static` class method that's called from the constructor and the *set* method.

🚫🐛 Error-Prevention Tip 10.1

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

```
 1  // Fig. 10.5: SalariedEmployee.java
 2  // SalariedEmployee concrete class extends abstra
 3
 4  public class SalariedEmployee extends Employee {
 5     private double weeklySalary;
 6
 7     // constructor
 8     public SalariedEmployee(String firstName, Stri
 9        String socialSecurityNumber, double weeklyS
10        super(firstName, lastName, socialSecurityNu
11
12        if (weeklySalary < 0.0) {
```

```
13              throw new IllegalArgumentException(
14                  "Weekly salary must be >= 0.0");
15          }
16
17          this.weeklySalary = weeklySalary;
18      }
19
20      // set salary
21      public void setWeeklySalary(double weeklySalar
22          if (weeklySalary < 0.0) {
23              throw new IllegalArgumentException(
24                  "Weekly salary must be >= 0.0");
25          }
26
27          this.weeklySalary = weeklySalary;
28      }
29
30      // return salary
31      public double getWeeklySalary() {return weekly
32
33      // calculate earnings; override abstract metho
34      @Override
35      public double earnings() {return getWeeklySala
36
37      // return String representation of SalariedEmp
38      @Override
39      public String toString() {
40          return String.format("salaried employee: %s
41              super.toString(), "weekly salary", getWee
42      }
43  }
```

Fig. 10.5

SalariedEmployee concrete class extends abstract

class `Employee`.

Method `earnings` overrides `Employee`'s *abstract* method `earnings` to provide a *concrete* implementation that returns the `SalariedEmployee`'s weekly salary. If we do not implement `earnings`, class `SalariedEmployee` must be declared `abstract`—otherwise, class `SalariedEmployee` will not compile. Of course, we want `SalariedEmployee` to be a *concrete* class in this example.

Method `toString` (lines 38–42) overrides `Employee`'s `toString`. If class `SalariedEmployee` did *not* override `toString`, it would have inherited `Employee`'s version. In that case, `SalariedEmployee`'s `toString` method would simply return the employee's full name and social security number, which does *not* adequately represent a `SalariedEmployee`. To produce a complete `String` representation of a `SalariedEmployee`, the subclass's `toString` method returns `"salaried employee: "` followed by the superclass `Employee`-specific information (i.e., first name, last name and social security number) obtained by invoking the *superclass's* `toString` method (line 41)—this is a nice example of *code reuse*. The `String` representation of a `SalariedEmployee` also contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` method.

10.5.3 Concrete Subclass

HourlyEmployee

Class `HourlyEmployee` (Fig. 10.6) also extends
`Employee` (line 4). The class includes a constructor (lines 9–
24) that receives a first name, a last name, a social security
number, an hourly wage and the number of hours worked.
Lines 27–33 and 39–46 declare *set* methods that assign new
values to instance variables `wage` and `hours`, respectively.
Method `setWage` ensures that `wage` is *nonnegative,* and
method `setHours` ensures that the value of `hours` is
between 0 and 168 (the total number of hours in a week)
inclusive. Class `HourlyEmployee` also includes *get*
methods (lines 36 and 49) to return the values of `wage` and
`hours`, respectively; a method `earnings` (lines 52–60) to
calculate an `HourlyEmployee`'s earnings; and a method
`toString` (lines 63–68), which returns a `String`
containing the employee's type (`"hourly employee: "`)
and the employee-specific information. The
`HourlyEmployee` constructor, like the
`SalariedEmployee` constructor, passes the first name, last
name and social security number to the superclass `Employee`
constructor (line 11) to initialize the `private` instance
variables. In addition, method `toString` calls *superclass*
method `to-String` (line 66) to obtain the `Employee`-
specific information (i.e., first name, last name and social
security number)—this is another nice example of *code reuse*.

```
 1  // Fig. 10.6: HourlyEmployee.java
 2  // HourlyEmployee class extends Employee.
 3
```

```java
 4  public class HourlyEmployee extends Employee {
 5      private double wage; // wage per hour
 6      private double hours; // hours worked for week
 7
 8      // constructor
 9      public HourlyEmployee(String firstName, String
10          String socialSecurityNumber, double wage, d
11          super(firstName, lastName, socialSecurityNu
12
13          if (wage < 0.0) { // validate wage
14              throw new IllegalArgumentException("Hour
15          }
16
17          if ((hours < 0.0) || (hours > 168.0)) { //
18              throw new IllegalArgumentException(
19                  "Hours worked must be >= 0.0 and <=
20          }
21
22          this.wage = wage;
23          this.hours = hours;
24      }
25
26      // set wage
27      public void setWage(double wage) {
28          if (wage < 0.0) { // validate wage
29              throw new IllegalArgumentException("Hour
30          }
31
32          this.wage = wage;
33      }
34
35      // return wage
36      public double getWage() {return wage;}
37
38      // set hours worked
39      public void setHours(double hours) {
40          if ((hours < 0.0) || (hours > 168.0)) { //
41              throw new IllegalArgumentException(
42                  "Hours worked must be >= 0.0 and <= 1
43          }
```

```
                              44
         45              this.hours = hours;
                   46        }
                      47
         48      // return hours worked
   49     public double getHours() {return hours;}
                      50
     51    // calculate earnings; override abstract metho
                   52      @Override
         53      public double earnings() {
   54       if (getHours() <= 40) { // no overtime
      55           return getWage() * getHours();
                   56           }
                57         else {
   58           return 40 * getWage() + (getHours() - 40
                   59           }
                   60       }
                      61
   62     // return String representation of HourlyEmplo
                   63      @Override
       64      public String toString() {
   65       return String.format("hourly employee: %s%n
   66           super.toString(), "hourly wage", getWage
      67            "hours worked", getHours());
                   68       }
                 69   }
```
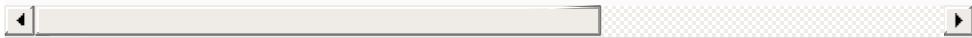
Fig. 10.6

HourlyEmployee class extends Employee.

10.5.4 Concrete Subclass

CommissionEmployee

Class `CommissionEmployee` ([Fig. 10.7](#)) extends class `Employee` (line 4). The class includes a constructor (lines 9–25) that takes a first name, a last name, a social security number, a sales amount and a commission rate; *set* methods (lines 28–34 and 40–47) to assign valid new values to instance variables `grossSales` and `commissionRate`, respectively; *get* methods (lines 37 and 50) that retrieve the values of these instance variables; method `earnings` (lines 53–56) to calculate a `CommissionEmployee`'s earnings; and method `toString` (lines 59–65), which returns `"commission employee: "` followed by the employee's specific information. The constructor also passes the first name, last name and social security number to *superclass* `Employee`'s constructor (line 12) to initialize `Employee`'s `private` instance variables. Method `toString` calls *superclass* method `toString` (line 62) to obtain the `Employee`-specific information (i.e., first name, last name and social security number).

```
 1  // Fig. 10.7: CommissionEmployee.java
 2  // CommissionEmployee class extends Employee.
 3
 4  public class CommissionEmployee extends Employee
 5     private double grossSales; // gross weekly sal
 6     private double commissionRate; // commission p
 7
 8     // constructor
 9     public CommissionEmployee(String firstName, St
10        String socialSecurityNumber, double grossSa
11           double commissionRate) {
```

```java
12          super(firstName, lastName, socialSecurityNu
13
14      if (commissionRate <= 0.0 || commissionRate
15         throw new IllegalArgumentException(
16            "Commission rate must be > 0.0 and <
17                }
18
19      if (grossSales < 0.0) { // validate
20         throw new IllegalArgumentException("Gros
21                }
22
23         this.grossSales = grossSales;
24         this.commissionRate = commissionRate;
25            }
26
27      // set gross sales amount
28      public void setGrossSales(double grossSales)
29         if (grossSales < 0.0) { // validate
30            throw new IllegalArgumentException("Gro
31                }
32
33            this.grossSales = grossSales;
34            }
35
36      // return gross sales amount
37      public double getGrossSales() {return grossSa
38
39      // set commission rate
40      public void setCommissionRate(double commissi
41         if (commissionRate <= 0.0 || commissionRat
42            throw new IllegalArgumentException(
43               "Commission rate must be > 0.0 and <
44                }
45
46         this.commissionRate = commissionRate;
47            }
48
49      // return commission rate
50      public double getCommissionRate() {return com
51
```

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

Fig. 10.7

CommissionEmployee class extends Employee.

10.5.5 Indirect Concrete Subclass BasePlusCommissionEmployee

Class BasePlusCommissionEmployee (Fig. 10.8)
extends class CommissionEmployee (line 4) and therefore
is an *indirect* subclass of class Employee. Class

`BasePlusCommission-Employee` has a constructor (lines 8–19) that receives a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes all of these except the base salary to the `CommissionEmployee` constructor (lines 11–12) to initialize the superclass instance variables. `BasePlusCommissionEmployee` also contains a *set* method (lines 22–28) to assign a new value to instance variable `baseSalary` and a *get* method (line 31) to return `baseSalary`'s value. Method `earnings` (lines 34–35) calculates a `BasePlusCommissionEmployee`'s earnings. Line 35 in method `earnings` calls *superclass* `CommissionEmployee`'s `earnings` method to calculate the commission-based portion of the employee's earnings— this is another nice example of *code reuse*. `BasePlusCommissionEmployee`'s `toString` method (lines 38–43) creates a `String` representation of a `BasePlusCommissionEmployee` that contains `"base-salaried"`, followed by the `String` obtained by invoking *superclass* `CommissionEmployee`'s `toString` method (line 41), then the base salary. The result is a `String` beginning with `"base-salaried commission employee"` followed by the rest of the `BasePlusCommissionEmployee`'s information. Recall that `CommissionEmployee`'s `toString` obtains the employee's first name, last name and social security number by invoking the `toString` method of its *superclass* (i.e., `Employee`)—yet another example of *code reuse*. `BasePlusCommissionEmployee`'s `toString` method initiates a *chain of method calls* that span all three levels of the

Employee hierarchy.

```
 1  // Fig. 10.8: BasePlusCommissionEmployee.java
 2  // BasePlusCommissionEmployee class extends Commi
 3
 4  public class BasePlusCommissionEmployee extends C
 5     private double baseSalary; // base salary per
 6
 7     // constructor
 8     public BasePlusCommissionEmployee(String first
 9        String socialSecurityNumber, double grossSa
10        double commissionRate, double baseSalary) {
11        super(firstName, lastName, socialSecurityNu
12              grossSales, commissionRate);
13
14        if (baseSalary < 0.0) { // validate baseSal
15           throw new IllegalArgumentException("Base
16        }
17
18        this.baseSalary = baseSalary;
19     }
20
21     // set base salary
22     public void setBaseSalary(double baseSalary) {
23        if (baseSalary < 0.0) { // validate baseSal
24           throw new IllegalArgumentException("Base
25        }
26
27        this.baseSalary = baseSalary;
28     }
29
30     // return base salary
31     public double getBaseSalary() {return baseSala
32
33     // calculate earnings; override method earning
34     @Override
35     public double earnings() {return getBaseSalary
36
37     // return String representation of BasePlusCom
```

```
38      @Override
39    public String toString() {
40      return String.format("%s %s; %s: $%,.2f",
41          "base-salaried", super.toString(),
42          "base salary", getBaseSalary());
43      }
44  }
```

◄   ►
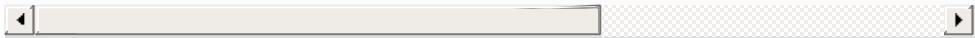
Fig. 10.8

`BasePlusCommissionEmployee` class extends
`CommissionEmployee`.

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting

To test our `Employee` hierarchy, the application in Fig. 10.9 creates an object of each of the four *concrete* classes `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`. The program 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%—this 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. Lines 7–16 create objects of each of the four concrete Employee subclasses. Lines 20–28 output the String representation and earnings of each of these objects *nonpolymorphically*. Each object's toString method is called *implicitly* by printf when the object is output as a String with the %s format specifier.

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest {
5    public static void main(String[] args) {
6        // create subclass objects
7        SalariedEmployee salariedEmployee =
8          new SalariedEmployee("John", "Smith", "1
9          HourlyEmployee hourlyEmployee =
10         new HourlyEmployee("Karen", "Price", "22
11        CommissionEmployee commissionEmployee =
12            new CommissionEmployee(
13          "Sue", "Jones", "333-33-3333", 10000, .0
14        BasePlusCommissionEmployee basePlusCommissi
15            new BasePlusCommissionEmployee(
16          "Bob", "Lewis", "444-44-4444", 5000, .04
17
18        System.out.println("Employees processed ind
19
20        System.out.printf("%n%s%n%s: $%,.2f%n%n",
21            salariedEmployee, "earned", salariedEmpl
22        System.out.printf("%s%n%s: $%,.2f%n%n",
23            hourlyEmployee, "earned", hourlyEmployee
24        System.out.printf("%s%n%s: $%,.2f%n%n",
25            commissionEmployee, "earned", commission
```

```java
26          System.out.printf("%s%n%s: $%,.2f%n%n",
   27              basePlusCommissionEmployee,
28          "earned", basePlusCommissionEmployee.ear
               29
30      // create four-element Employee array
31      Employee[] employees = new Employee[4];
               32
  33      // initialize array with Employees
   34       employees[0] = salariedEmployee;
    35        employees[1] = hourlyEmployee;
  36       employees[2] = commissionEmployee;
37      employees[3] = basePlusCommissionEmployee;
               38
39      System.out.printf("Employees processed poly
               40
41      // generically process each element in arra
42      for (Employee currentEmployee : employees)
43          System.out.println(currentEmployee); //
               44
45          // determine whether element is a BasePl
46          if (currentEmployee instanceof BasePlusC
47              // downcast Employee reference to
48              // BasePlusCommissionEmployee referen
49              BasePlusCommissionEmployee employee =
50                 (BasePlusCommissionEmployee) curre
               51
52              employee.setBaseSalary(1.10 * employe
               53
   54                 System.out.printf(
55              "new base salary with 10%% increas
  56                 employee.getBaseSalary());
            57          }
               58
      59            System.out.printf(
60          "earned $%,.2f%n%n", currentEmployee.
            61          }
               62
63      // get type name of each object in employee
64      for (int j = 0; j < employees.length; j++)
65          System.out.printf("Employee %d is a %s%n
```

```
66                    employees[j].getClass().getName());
67            }
68        }
69    }
```

```
         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 s
                  earned: $500.00

         Employees processed polymorphically:

            salaried employee: John Smith
         social security number: 111-11-1111
              weekly salary: $800.00
                  earned $800.00
```
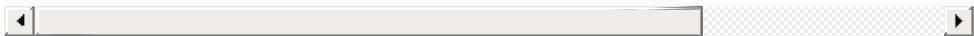
```
            hourly employee: Karen Price
         social security number: 222-22-2222
```

```
         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 s
     new base salary with 10% increase is: $330.00
                   earned $530.00

            Employee 0 is a SalariedEmployee
             Employee 1 is a HourlyEmployee
            Employee 2 is a CommissionEmployee
         Employee 3 is a BasePlusCommissionEmployee
```
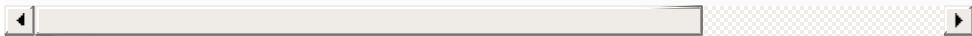
Fig. 10.9

`Employee` hierarchy test program.

Creating the Array of Employees

Line 31 declares `employees` and assigns it an array of four
`Employee` variables. Lines 34–37 assign to the elements a
reference to a `SalariedEmployee`, an
`HourlyEmployee`, a `CommissionEmployee` and a

`BasePlusCommissionEmployee`, respectively. These assignments are allowed, because a `SalariedEmployee` *is an* `Employee`, an `HourlyEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`. Therefore, we can assign the references of `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee` objects to *superclass* `Employee` variables, *even though* `Employee` *is an abstract class*.

Polymorphically Processing Employees

Lines 42–61 iterate through array `employees` and invoke methods `toString` and `earnings` with `Employee` variable `currentEmployee`, which is assigned the reference to a different `Employee` in the array on each iteration. The output illustrates that the specific methods for each class are indeed invoked. All calls to method `toString` and `earnings` are resolved at *execution* time, based on the *type* of the object to which `currentEmployee` refers. This process is known as **dynamic binding** or **late binding**. For example, line 43 *implicitly* invokes method `toString` of the object to which `currentEmployee` refers. As a result of *dynamic binding,* Java decides which class's `toString` method to call *at execution time rather than at compile time*.

Only the methods of class `Employee` can be called via an `Employee` variable (and `Employee`, of course, includes the methods of class `Object`). A superclass reference can be used to invoke only methods of the *superclass*—the *subclass* method implementations are invoked *polymorphically*.

Performing Type-Specific Operations on `BasePlusCommissionEmployees`

We perform special processing on `BasePlusCommissionEmployee` objects—as we encounter these objects at execution time, we increase their base salary by 10%. When processing objects *polymorphically*, we typically do not need to worry about the *specifics*, but to adjust the base salary, we *do* have to determine the *specific* type of `Employee` object at *execution time*. Line 46 uses the `instanceof` operator to determine whether a particular `Employee` object's type is `BasePlusCommissionEmployee`. The condition in line 46 is *true* if the object referenced by `currentEmployee` *is a* `BasePlusCommissionEmployee`. This would also be *true* for any object of a `BasePlusCommissionEmployee` subclass because of the *is-a* relationship a subclass has with its superclass. Lines 49–50 *downcast* `currentEmployee` from type `Employee` to type

`BasePlusCommissionEmployee`—this cast is allowed only if the object has an *is-a* relationship with `BasePlusCommissionEmployee`. The condition at line 46 ensures that this is the case. This cast is required if we're to invoke subclass `BasePlusCommissionEmployee` methods `getBaseSalary` and `setBaseSalary` on the current `Employee` object—as you'll see momentarily, *attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.*

Common Programming Error 10.3

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

Common Programming Error 10.4

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

If the `instanceof` expression in line 46 is `true`, lines 49–56 perform the special processing required for the

BasePlusCommissionEmployee object. Using
BasePlusCommissionEmployee variable employee,
line 52 invokes subclass-only methods getBaseSalary and
setBaseSalary to retrieve and update the employee's base
salary with the 10% raise.

Calling earnings Polymorphically

Lines 59–60 invoke method earnings on
currentEmployee, which polymorphically calls the
appropriate subclass object's earnings method. Obtaining
the earnings of the SalariedEmployee,
HourlyEmployee and CommissionEmployee
polymorphically in lines 59–60 produces the same results as
obtaining these employees' earnings individually in lines 20–
25. The earnings amount obtained for the
BasePlusCommissionEmployee in lines 59–60 is higher
than that obtained in lines 26–28, due to the 10% increase in
its base salary.

Getting Each Employee's Class Name

Lines 64–67 display each employee's type as a String.
Every object *knows its own class* and can access this

information through the `getClass` method, which all classes inherit from class `Object`. Method `getClass` returns an object of type `Class` (from package `java.lang`), which contains information about the object's type, including its class name. Line 66 invokes `getClass` on the current object to get its class. The result of the `getClass` call is used to invoke `getName` to get the object's class name.

Avoiding Compilation Errors with Downcasting

In the previous example, we avoided several compilation errors by *downcasting* an `Employee` variable to a `BasePlusCommissionEmployee` variable in lines 49–50. If you remove the cast operator `(BasePlusCommissionEmployee)` from line 50 and attempt to assign `Employee` variable `currentEmployee` directly to `BasePlusCommissionEmployee` variable `employee`, you'll receive an "`incompatible types`" compilation error. This error indicates that the attempt to assign the reference of superclass object `currentEmployee` to subclass variable `employee` is *not* allowed. The compiler prevents this assignment because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`—*the* is-a *relationship applies only between the subclass and its superclasses, not vice versa.*

Similarly, if lines 52 and 56 used superclass variable
`currentEmployee` to invoke subclass-only methods
`getBaseSalary` and `setBaseSalary`, we'd receive
"`cannot find symbol`" compilation errors at these lines.
Attempting to invoke subclass-only methods via a superclass
variable is *not* allowed—even though lines 52 and 56 execute
only if `instanceof` in line 46 returns `true` to indicate that
`currentEmployee` holds a reference to a
`BasePlusCommissionEmployee` object. Using a
superclass `Employee` variable, we can invoke only methods
found in class `Employee`—`earnings`, `toString` and
`Employee`'s *get* and *set* methods.

Software Engineering Observation 10.5

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