

Polymorphism2

Objectives

In this chapter you will learn:

- The concept of polymorphism.
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.

Polymorphism

- Polymorphism
 - Enables “programming in the general”
 - The same invocation can produce “many forms” of results
 - When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable
 - The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked
 - Facilitates adding new classes to a system with minimal modifications to the system’s code

Software Engineering Observation

- Polymorphism enables programmers to deal in generalities and let the execution-time environment handle the specifics. Programmers can command objects to behave in manners appropriate to those objects, without knowing the types of the objects (as long as the objects belong to the same inheritance hierarchy).
- Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the base system. Only client code that instantiates new objects must be modified to accommodate new types.

Demonstrating Polymorphic Behavior

- A superclass reference can be aimed at a subclass object
 - This is possible because a subclass object *is a* superclass object as well
 - When invoking a method from that reference, the type of the actual referenced object, not the type of the reference, determines which method is called
- A subclass reference can be aimed at a superclass object only if the object is downcasted

Employee Case Study

```
1 // Fig. 9.12: CommissionEmployee3.java
2 // CommissionEmployee3 class represents a commission employee.
3
4 public class CommissionEmployee3
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 CommissionEmployee3( 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 CommissionEmployee3 constructor
23
24    // set first name
25    public void setFirstName( String first )
26    {
27        firstName = first;
28    } // end method setFirstName
29
```

Declare private
instance variables

- CommissionEmployee3.java

```
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;
40 } // end method setLastName
41
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
```

```
60 // set gross sales amount
61 public void setGrossSales( double sales )
62 {
63     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64 } // end method setGrossSales
65
66 // return gross sales amount
67 public double getGrossSales()
68 {
69     return grossSales;
70 } // end method getGrossSales
71
72 // set commission rate
73 public void setCommissionRate( double rate )
74 {
75     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76 } // end method setCommissionRate
77
78 // return commission rate
79 public double getCommissionRate()
80 {
81     return commissionRate;
82 } // end method getCommissionRate
83
```



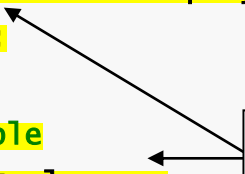
```
84 // calculate earnings
85 public double earnings()
86 {
87     return getCommissionRate() * getGrossSales();
88 } // end method earnings
89
90 // return String representation of CommissionEmployee3 object
91 public String toString()
92 {
93     return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94         "commission employee", getFirstName(), getLastName(),
95         "social security number", getSocialSecurityNumber(),
96         "gross sales", getGrossSales(),
97         "commission rate", getCommissionRate() );
98 } // end method toString
99 } // end class CommissionEmployee3
```

```

1 // Fig. 9.13: BasePlusCommissionEmployee4.java
2 // BasePlusCommissionEmployee4 class inherits from CommissionEmployee3 and
3 // accesses CommissionEmployee3's private data via CommissionEmployee3's
4 // public methods.
5
6 public class BasePlusCommissionEmployee4 extends CommissionEmployee3
7 {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee4( 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 BasePlusCommissionEmployee4 constructor
17
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21        baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // end method setBaseSalary
23
24    // return base salary
25    public double getBaseSalary()
26    {
27        return baseSalary;
28    } // end method getBaseSalary
29
30    // calculate earnings
31    public double earnings()
32    {
33        return getBaseSalary() + super.earnings();
34    } // end method earnings
35
36    // return String representation of BasePlusCommissionEmployee4
37    public String toString()
38    {
39        return String.format( "%s %s\n%s: %.2f", "base-salaried",
40            super.toString(), "base salary", getBaseSalary() );
41    } // end method toString
42 } // end class BasePlusCommissionEmployee4

```

```
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        CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
11            "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13        // assign subclass reference to subclass variable
14        BasePlusCommissionEmployee4 basePlusCommissionEmployee =
15            new BasePlusCommissionEmployee4(
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 CommissionEmployee3's toString with superclass reference ",
21            "to superclass object", commissionEmployee.toString() );
22
23        // invoke toString on subclass object using subclass variable
24        System.out.printf( "%s %s:\n\n%s\n\n",
25            "Call BasePlusCommissionEmployee4's toString with subclass",
26            "reference to subclass object",
27            basePlusCommissionEmployee.toString() );
28
```



Typical reference
assignments

```

29      // invoke toString on subclass object using superclass variable
30      CommissionEmployee3 commissionEmployee2 =
31          basePlusCommissionEmployee;
32      System.out.printf( "%s %s:\n\n%s\n",
33          "Call BasePlusCommissionEmployee4's toString with superclass",
34          "reference to subclass object", commissionEmployee2.toString() );
35  } // end main
36 } // end class PolymorphismTest

```

Assign a reference to a **basePlusCommissionEmployee** object to a **CommissionEmployee3** variable

Call **CommissionEmployee3**'s **toString** with superclass reference to superclass object:

```

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

```

Call **BasePlusCommissionEmployee4**'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 **BasePlusCommissionEmployee4**'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

```

Polymorphically call **basePlusCommissionEmployee**'s **toString** method

Abstract Class in Java

A class that is declared using “**abstract**” keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods.

An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it.

Why we need an abstract class?

Lets say we have a class **Animal** that has a method `sound()` and the **subclasses** of it like **Dog, Lion, Horse, Cat** etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

So when we know that all the animal **child classes will and should override** this method, then there is no point to implement this method in parent class. Thus, making this **method abstract** would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, we must declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

Examples

```
//abstract parent class
abstract class Animal{

//abstract method
    public abstract void sound();
}
```

```
//Dog class extends Animal class
public class Dog extends Animal{
    public void sound(){
        System.out.println("Woof"); }
}
```

```
//Lion class extends Animal class
public class Lion extends Animal{
    public void sound(){
        System.out.println("Roar"); }
}
```

```
public static void main(String [] args)
{
    Dog obj1 = new Dog();
    Lion obj2=new Lion();
    obj1.sound(); obj2.sound;
}
```


Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration using abstract keyword
abstract class A{
    //This is abstract method
    abstract void myMethod();

    //This is concrete method with body
    void anotherMethod(){
        //Does something
    }
}
```

Rules for Abstraction:

Note 1: As we seen in the above example, there are **cases** when it is difficult or often **unnecessary to implement all the methods** in parent class. In these cases, we can declare the **parent class as abstract**, which makes it a special class which is **not complete on its own**.

A concrete class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Note 2: **Abstract class cannot be instantiated** which means you cannot create the object of it. To use this class, you need to create another class that extends this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of abstract parent class, then the child class must be declared abstract as well.

Abstract class vs Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, Animal is a abstract class and Cat, Dog & Lion are concrete classes.

Key Points:

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well.

Some basics of abstract method

- Abstract method has no body.
- Always end the declaration with a **semicolon(;)** .
- It must be **overridden**. An abstract class must be extended and in a same way abstract method must be overridden.
- A class has to be declared abstract to have abstract methods.

Example of Abstract class and method

```
abstract class MyClass{
    public void disp(){
        System.out.println("Concrete method of parent class");
    }
    abstract public void disp2();
}

class Demo extends MyClass{
    /* Must Override this method while extending
     * MyClas
     */
    public void disp2()
    {
        System.out.println("overriding abstract method");
    }
    public static void main(String args[]){
        Demo obj = new Demo();
        obj.disp2();
    }
}
```


Creating Abstract Superclass Employee

- `abstract superclass Employee`
 - `earnings` is declared `abstract`
 - No implementation can be given for `earnings` in the `Employee` abstract class
 - An array of `Employee` variables will store references to subclass objects
 - `earnings` method calls from these variables will call the appropriate version of the `earnings` method

Polymorphic information for the Employee hierarchy classes.

	earnings	toString
Employee	abstract	<i>firstName lastName</i> social security number: <i>SSN</i>
Salaried- Employee	weeklySalary	salaried employee: <i>firstName lastName</i> social security number: <i>SSN</i> weekly salary: <i>weeklySalary</i>
Hourly- Employee	<i>If hours <= 40</i> <i>wage * hours</i> <i>If hours > 40</i> <i>40 * wage +</i> <i>(hours - 40) * wage * 1.5</i>	hourly employee: <i>firstName lastName</i> social security number: <i>SSN</i> hourly wage: <i>wage</i> ; hours worked: <i>hours</i>
Commission- Employee	<i>commissionRate * grossSales</i>	commission employee: <i>firstName lastName</i> social security number: <i>SSN</i> gross sales: <i>grossSales</i> ; commission rate: <i>commissionRate</i>
BasePlus- Commission- Employee	<i>(commissionRate * grossSales) + baseSalary</i>	base salaried commission employee: <i>firstName lastName</i> social security number: <i>SSN</i> gross sales: <i>grossSales</i> ; commission rate: <i>commissionRate</i> ; base salary: <i>baseSalary</i>


```
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
```

Declare **abstract** class
Employee

Attributes common to all
employees

```
18 // set first name
19 public void setFirstName( String first )
20 {
21     firstName = first;
22 } // end method setFirstName
23
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;
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
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 public String toString()
56 {
57     return String.format( "%s %s\nsocial security number: %s",
58         getFirstName(), getLastName(), getSocialSecurityNumber() );
59 } // end method toString
60
61 // abstract method overridden by subclasses
62 public abstract double earnings(); // no implementation here
63 } // end abstract class Employee
```

abstract method
earnings has no
implementation

Subclass: Salaried Employee

```
1 // Fig. 10.5: SalariedEmployee.java
2 // SalariedEmployee class extends Employee.
```

```
3
4 public class SalariedEmployee extends Employee
5 {
```

Class **SalariedEmployee**
extends class **Employee**

```
6     private double weeklySalary;
```

```
7
```

```
8     // four-argument constructor
```

```
9     public SalariedEmployee( String first, String last, String ssn,
10         double salary )
```

Call superclass
constructor

```
11 {
```

```
12     super( first, last, ssn ); // pass to Employee constructor
```

```
13     setWeeklySalary( salary ); // validate and store salary
```

```
14 } // end four-argument SalariedEmployee constructor
```

```
15
```

```
16 // set salary
```

```
17 public void setWeeklySalary( double salary )
```

Call **setWeeklySalary**
method

```
18 {
```

```
19     weeklySalary = salary < 0.0 ? 0.0 : salary;
```

```
20 } // end method setWeeklySalary
```

Validate and set weekly salary
value

```
21
```

```
22 // return salary
23 public double getWeeklySalary()
24 {
25     return weeklySalary;
26 } // end method getWeeklySalary
27
28 // calculate earnings; override abstract method earnings in Employee
29 public double earnings()
30 {
31     return getWeeklySalary();
32 } // end method earnings
33
34 // return String representation of SalariedEmployee object
35 public String toString()
36 {
37     return String.format( "salaried employee: %s\n%s: $%,.2f",
38         super.toString(), "weekly salary", getWeeklySalary() );
39 } // end method toString
40 } // end class SalariedEmployee
```

Override **earnings** method so **SalariedEmployee** can be concrete

Override **toString** method

Call superclass's version of **toString**

Subclass: Hourly Employee

```
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
18    // set wage
19    public void setWage( double hourlywage )
20    {
21        wage = ( hourlywage < 0.0 ) ? 0.0 : hourlywage;
22    } // end method setWage
23
24    // return wage
25    public double getWage()
26    {
27        return wage;
28    } // end method getWage
29
```

Class **HourlyEmployee** extends
class **Employee**

Call superclass constructor

Validate and set hourly wage
value

```

30 // set hours worked
31 public void setHours( double hoursworked )
32 {
33     hours = ( ( hoursworked >= 0.0 ) && ( hoursworked <= 168.0 ) ) ?
34         hoursworked : 0.0;
35 } // end method setHours
36
37 // return hours worked
38 public double getHours()
39 {
40     return hours;
41 } // end method getHours
42
43 // calculate earnings; override abstract method earnings in Employee
44 public double earnings()
45 {
46     if ( getHours() <= 40 ) // no overtime
47         return getWage() * getHours();
48     else
49         return 40 * getWage() + ( getHours() - 40 ) * getWage() * 1.5;
50 } // end method earnings
51
52 // return String representation of HourlyEmployee object
53 public String toString()
54 {
55     return String.format( "hourly employee: %s\n%s: $%,.2f; %s: $%,.2f",
56         super.toString(), "hourly wage", getWage(),
57         "hours worked", getHours() );
58 } // end method toString
59 } // end class HourlyEmployee

```

Validate and set hours worked
value

Override **earnings** method so
HourlyEmployee can be concrete

Override **toString** method

Call superclass's **toString**
method

Subclass: Commission Employee

```
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
18    // set commission rate
19    public void setCommissionRate( double rate )
20    {
21        commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22    } // end method setCommissionRate
23
```

Class **CommissionEmployee**
extends class **Employee**

Call superclass
constructor

Validate and set commission rate
value

```

24 // return commission rate
25 public double getCommissionRate()
26 {
27     return commissionRate;
28 } // end method getCommissionRate
29
30 // set gross sales amount
31 public void setGrossSales( double sales )
32 {
33     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34 } // end method setGrossSales
35
36 // return gross sales amount
37 public double getGrossSales()
38 {
39     return grossSales;
40 } // end method getGrossSales
41

```

Override **earnings** method so
CommissionEmployee can be
concrete

```

42 // calculate earnings; override abstract method earnings in Employee
43 public double earnings()
44 {
45     return getCommissionRate() * getGrossSales();
46 } // end method earnings
47
48 // return String representation of CommissionEmployee object
49 public String toString()
50 {
51     return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
52         "commission employee", super.toString(),
53         "gross sales", getGrossSales(),
54         "commission rate", getCommissionRate() );
55 } // end method toString
56 } // end class CommissionEmployee

```


Subclass: BasePlusCommission Employee

```
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
16     // set base salary
17     public void setBaseSalary( double salary )
18     {
19         baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
20     } // end method setBaseSalary
21
```

Class **BasePlusCommissionEmployee**
extends class **CommissionEmployee**

Call superclass constructor

Validate and set base salary
value

```
22 // return base salary
23 public double getBaseSalary()
24 {
25     return baseSalary;
26 } // end method getBaseSalary
27
28 // calculate earnings; override method earnings in CommissionEmployee
29 public double earnings()
30 {
31     return getBaseSalary() + super.earnings();
32 } // end method earnings
33
34 // return String representation of BasePlusCommissionEmployee object
35 public String toString()
36 {
37     return String.format( "%s %s; %s: $%,.2f",
38         "base-salaried", super.toString(),
39         "base salary", getBaseSalary() );
40 } // end method toString
41 } // end class BasePlusCommissionEmployee
```

Override **earnings** method

Call superclass's **earnings**
method

Override **toString** method

Call superclass's **toString**
method

Payroll System Test Application:

```
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
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
43 // generically process each element in array employees
44 for ( Employee currentEmployee : employees )
45 {
46     System.out.println( currentEmployee ); // invokes toString
47 }
```

Assigning subclass objects to superclass variables

Implicitly and polymorphically call
toString

```

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     double oldBaseSalary = employee.getBaseSalary();
57     employee.setBaseSalary( 1.10 * oldBaseSalary );
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
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

```

If the **currentEmployee** variable points to a **BasePlusCommissionEmployee** object

Downcast **currentEmployee** to a **BasePlusCommissionEmployee** reference

Give **BasePlusCommissionEmployees** a 10% base salary bonus

Polymorphically call **earnings** method

Call **getClass** and **getName** methods to display each **Employee** subclass object's class name

Output:

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

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

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

Same results as when the
employees were processed
individually

Base salary is increased by
10%

Each employee's type is
displayed