Chapter 10 Object-Oriented Programming: Polymorphism

Java™ How to Program, 9/e

OBJECTIVES

In this Chapter you'll 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.
- How polymorphism makes systems extensible and maintainable.
- To determine an object's type at execution time.
- To declare and implement interfaces.

10.1 Introduction

- Polymorphism
  - Enables you to “program in the general” rather than “program in the specific.”
  - Polymorphism enables you to write programs that process objects that share the same superclass as if they’re all objects of the superclass; this can simplify programming.

10.1 Introduction (Cont.)

- Example: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation.
  - Each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x*-*y* coordinates. Each subclass implements method `move`.
  - A program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per second—namely, `move`.

10.1 Introduction (Cont.)

- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same message (i.e., `move`) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

10.1 Introduction (Cont.)

- With polymorphism, we can design and implement systems that are easily *extensible*
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

10.1 Introduction (Cont.)

- Once a class implements an **interface**, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface.
- This is true of all subclasses of that class as well.
- Interfaces are particularly useful for assigning common functionality to possibly unrelated classes.
  - Allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls.

10.1 Introduction (Cont.)

- 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.

10.2 Polymorphism Examples

- Example: Quadrilaterals
  - If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
  - Any operation that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`.
  - These operations can also be performed on other `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.
  - Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

10.2 Polymorphism Examples (Cont.)

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

10.2 Polymorphism Examples (Cont.)

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

Software Engineering Observation 10.1

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

Software Engineering Observation 10.2

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

10.3 Demonstrating Polymorphic Behavior

- In the next example, we aim a superclass reference at a subclass object.
  - Invoking a method on a subclass object via a superclass reference invokes the subclass functionality
  - The type of the referenced object, not the type of the variable, determines which method is called
- This example demonstrates that an object of a subclass can be treated as an object of its superclass, enabling various interesting manipulations.
- A program can create an array of superclass variables that refer to objects of many subclass types.
  - Allowed because each subclass object *is an* object of its superclass.

10.3 Demonstrating Polymorphic Behavior (Cont.)

- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.

```
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.)

10.3 Demonstrating Polymorphic Behavior (Cont.)

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.
  - The Java compiler allows this "crossover" because an object of a subclass *is an object of its superclass (but not vice versa).*
- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type.
  - If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use.
  - This process is called dynamic binding.

10.4 Abstract Classes and Methods

- Abstract classes
  - Sometimes it's useful to declare classes for which you never intend to create objects.
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called abstract superclasses.
  - Cannot be used to instantiate objects—abstract classes are incomplete.
  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

10.4 Abstract Classes and Methods (Cont.)

- Classes that can be used to instantiate objects are called concrete classes.
- Such classes provide implementations of every method they declare (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.

10.4 Abstract Classes and Methods (Cont.)

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

10.4 Abstract Classes and Methods (Cont.)

- You make a class abstract by declaring it with keyword abstract.
- An abstract class normally contains one or more abstract methods.
  - An abstract method is 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`.

Software Engineering Observation 10.3

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

Common Programming Error 10.1

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

Common Programming Error 10.2

Failure to implement a superclass's abstract methods in a subclass is a compilation error unless the subclass is also declared `abstract`.

10.4 Abstract Classes and Methods (Cont.)

- Cannot instantiate objects of abstract superclasses, but you can use abstract superclasses to declare variables
  - These can hold references to objects of any concrete class derived from those abstract superclasses.
  - Programs typically use such variables to manipulate subclass objects polymorphically.
- Can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

10.4 Abstract Classes and Methods (Cont.)

- Polymorphism is particularly effective for implementing so-called *layered software systems*.
- Example: Operating systems and device drivers.
  - Commands to read or write data from and to devices may have a certain uniformity.
  - Device drivers control all communication between the operating system and the devices.
  - A write message sent to a device-driver object is interpreted in the context of that driver and how it manipulates devices of a specific type.
  - The write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device.

10.4 Abstract Classes and Methods (Cont.)

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

10.5 Case Study: Payroll System Using Polymorphism

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

10.5 Case Study: Payroll System Using Polymorphism (Cont.)

- `abstract` class `Employee` represents the general concept of an employee.
- Subclasses: `SalariedEmployee`, `CommissionEmployee`, `HourlyEmployee` and `BasePlusCommissionEmployee` (an indirect subclass)
- Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application.
- Abstract class names are italicized in the UML.

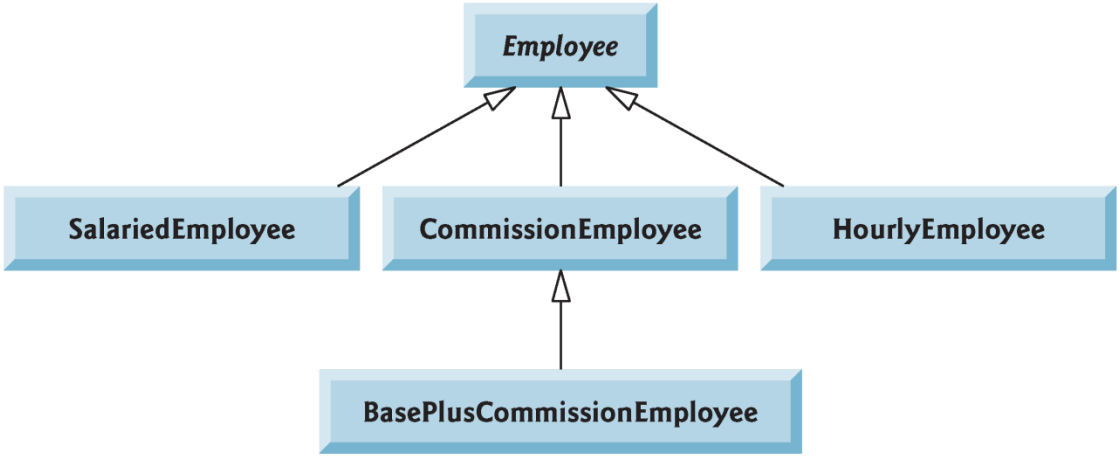

Fig. 10.2 | `Employee` hierarchy UML class diagram.

10.5 Case Study: Payroll System Using Polymorphism (Cont.)

- Abstract superclass `Employee` declares the "interface" to the hierarchy—that is, the set of methods that a program can invoke on all `Employee` objects.
  - We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any `Employee` subclass.
- Each employee has a first name, a last name and a social security number defined in abstract superclass `Employee`.

10.5.1 Abstract Superclass Employee

- Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString`, in addition to the *get* and *set* methods that manipulate `Employee`'s instance variables.
- An `earnings` method applies to all employees, but each earnings calculation depends on the employee's class.
  - An `abstract` method—there is not enough information to determine what amount `earnings` should return.
  - Each subclass overrides `earnings` with an appropriate implementation.
- Iterate through the array of `Employee`s and call method `earnings` for each `Employee` subclass object.
  - Method calls processed polymorphically.

10.5.1 Abstract Superclass `Employee` (Cont.)

- The diagram in Fig. 10.3 shows each of the five classes in the hierarchy down the left side and methods `earnings` and `toString` across the top.
- For each class, the diagram shows the desired results of each method.
- Declaring the `earnings` method `abstract` indicates that each concrete subclass must provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*

Hourly-Employee	if (hours <= 40)
	  wage * hours
	else if (hours > 40)
	{
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee
5  {
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.)

10.5.2 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.)

10.5.3 Concrete Subclass HourlyEmployee

```
1  // Fig. 10.6: HourlyEmployee.java
2  // HourlyEmployee class extends Employee.
3
4  public class HourlyEmployee extends Employee
5  {
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.)

10.5.4 Concrete Subclass `CommissionEmployee`

```
1  // Fig. 10.7: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3
4  public class CommissionEmployee extends Employee
5  {
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.)

10.5.5 Indirect Concrete Subclass `BasePlusCommissionEmployee`

```
1  // Fig. 10.8: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class extends CommissionEmployee.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
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.)

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting

- Fig. 10.9 creates an object of each of the four concrete.
  - Manipulates these objects nonpolymorphically, via variables of each object's own type, then polymorphically, using an array of `Employee` variables.
- While processing the objects polymorphically, the program increases the base salary of each `BasePlusCommissionEmployee` by 10%
  - Requires determining the object's type at execution time.
- Finally, the program polymorphically determines and outputs the type of each object in the `Employee` array.

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest
5  {
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.)