9

Object-Oriented Programming: Inheritance

◀ ▶

Say not you know another entirely, till you have divided an inheritance with him.

— Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.

— Bertrand Russell

Good as it is to inherit a library, it is better to collect one.

— Augustine Birrell

Save base authority from others' books.

— William Shakespeare

OBJECTIVES

In this chapter you will learn:

- How inheritance promotes software reusability.
- The notions of superclasses and subclasses.
- To use keyword `extends` to create a class that inherits attributes and behaviors from another class.
- To use access modifier `protected` to give subclass methods access to superclass members.
- To access superclass members with `super`.
- How constructors are used in inheritance hierarchies.
- The methods of class `Object`, the direct or indirect superclass of all classes in Java.

Outline

9.1 Introduction

9.2 Superclasses and Subclasses

9.3 `protected` Members

9.4 Relationship between Superclasses and Subclasses

- 9.4.1 Creating and Using a `CommissionEmployee` Class
- 9.4.2 Creating a `BasePlusCommissionEmployee` Class without Using Inheritance
- 9.4.3 Creating a `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy
- 9.4.4 `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy Using `protected` Instance Variables
- 9.4.5 `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables

Outline

9.5 Constructors in Subclasses

9.6 Software Engineering with Inheritance

9.7 `Object` Class

9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels

9.9 Wrap-Up

9.1 Introduction

- Inheritance
  - Software reusability
  - Create new class from existing class
    - Absorb existing class’s data and behaviors
    - Enhance with new capabilities
  - Subclass extends superclass
    - Subclass
      - More specialized group of objects
      - Behaviors inherited from superclass
        - Can customize
      - Additional behaviors

9.1 Introduction (Cont.)

- Class hierarchy
  - Direct superclass
    - Inherited explicitly (one level up hierarchy)
  - Indirect superclass
    - Inherited two or more levels up hierarchy
  - Single inheritance
    - Inherits from one superclass
  - Multiple inheritance
    - Inherits from multiple superclasses
      - Java does not support multiple inheritance

9.2 Superclasses and subclasses

- **Superclasses and subclasses**
  - **Object of one class “is an” object of another class**
    - **Example: Rectangle is quadrilateral.**
      - **Class `Rectangle` inherits from class `Quadrilateral`**
      - **`Quadrilateral`: superclass**
      - **`Rectangle`: subclass**
  - **Superclass typically represents larger set of objects than subclasses**
    - **Example:**
      - **superclass: `Vehicle`**
        - **Cars, trucks, boats, bicycles, …**
      - **subclass: `Car`**
        - **Smaller, more-specific subset of vehicles**

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

Fig. 9.1 | **Inheritance examples.**

9.2 Superclasses and subclasses (Cont.)

- **Inheritance hierarchy**
  - **Inheritance relationships: tree-like hierarchy structure**
  - **Each class becomes**
    - **superclass**
      - **Supply members to other classes**

    OR
    - **subclass**
      - **Inherit members from other classes**

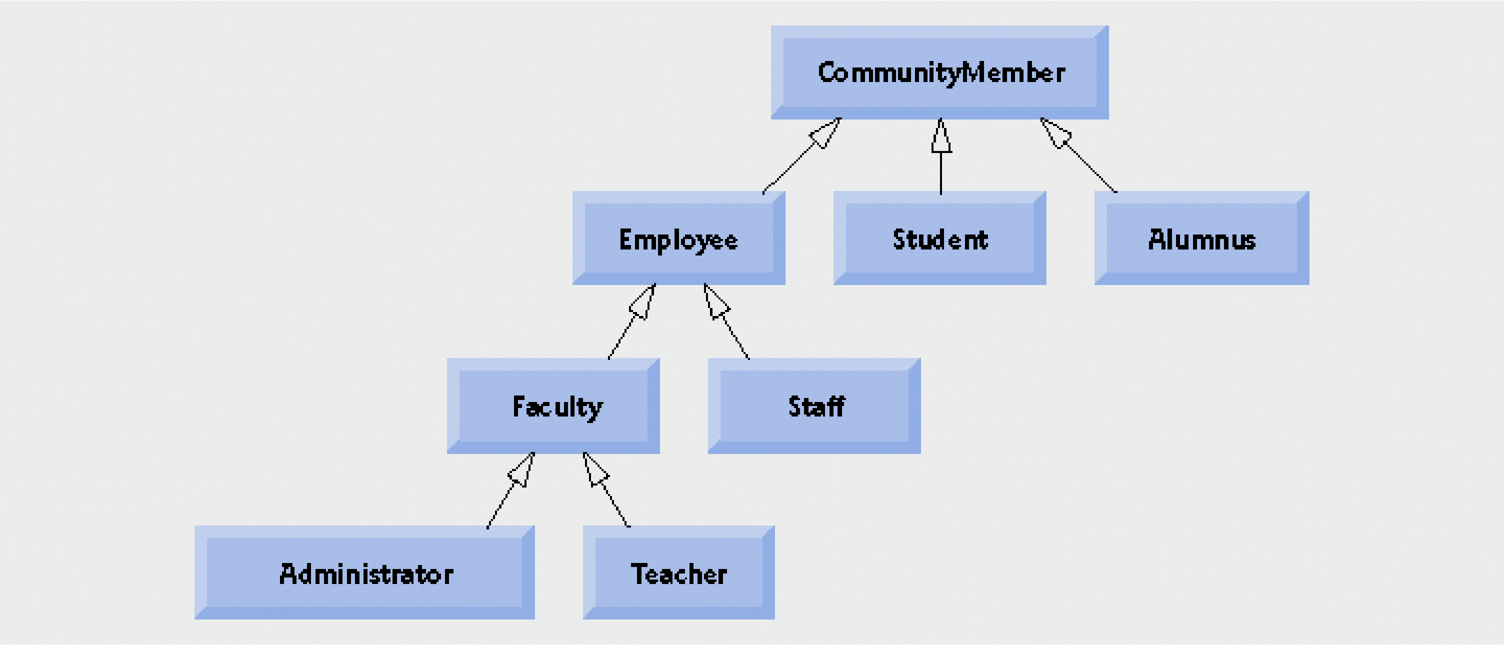


Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`

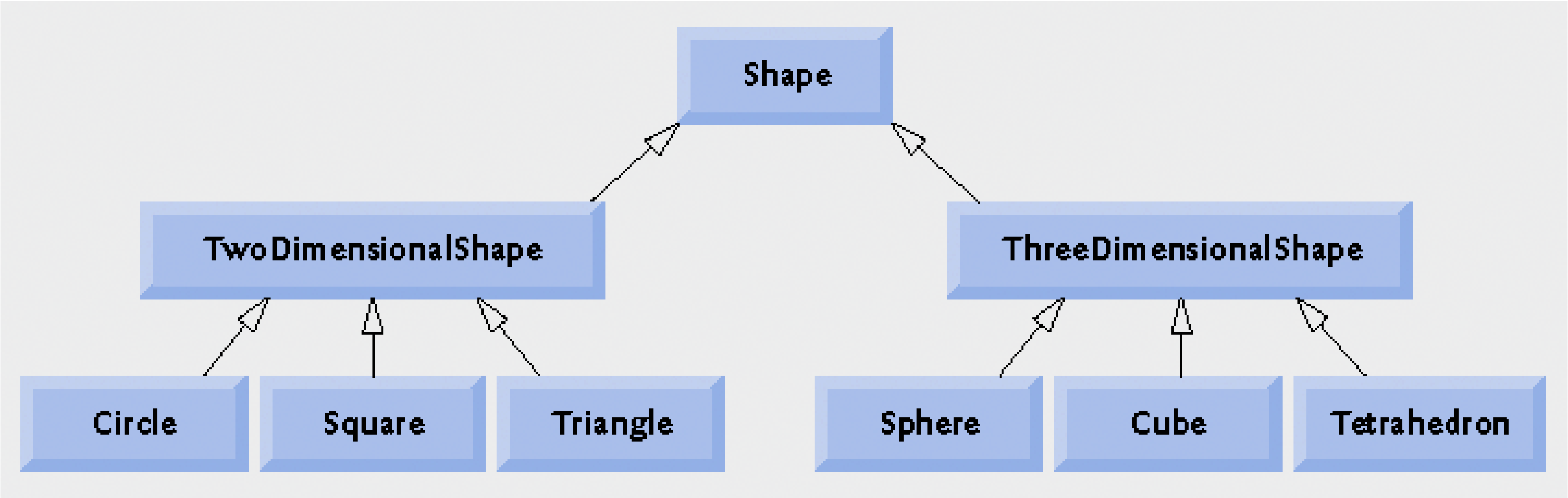


Fig. 9.3 | **Inheritance hierarchy for Shapes.**

9.3 `protected` Members

- `protected` access
  - Intermediate level of protection between `public` and `private`
  - `protected` members accessible by
    - superclass members
    - subclass members
    - Class members in the same package
  - Subclass access to superclass member
    - Keyword `super` and a dot (.)

Software Engineering Observation 9.1

Methods of a subclass cannot directly access `private` members of their superclass. A subclass can change the state of `private` superclass instance variables only through non-`private` methods provided in the superclass and inherited by the subclass.

Software Engineering Observation 9.2

Declaring `private` instance variables helps programmers test, debug and correctly modify systems. If a subclass could access its superclass's `private` instance variables, classes that inherit from that subclass could access the instance variables as well. This would propagate access to what should be `private` instance variables, and the benefits of information hiding would be lost.

9.4 Relationship between Superclasses and Subclasses

- **Superclass and subclass relationship**
  - **Example:** `CommissionEmployee/BasePlusCommissionEmployee` **inheritance hierarchy**
    - `CommissionEmployee`
      - **First name, last name, SSN, commission rate, gross sale amount**
    - `BasePlusCommissionEmployee`
      - **First name, last name, SSN, commission rate, gross sale amount**
      - **Base salary**

9.4.1 Creating and Using a CommissionEmployee Class

- Class CommissionEmployee
  - Extends class Object
    - Keyword extends
    - Every class in Java extends an existing class
      - Except Object
    - Every class inherits Object’s methods
    - New class implicitly extends Object
      - If it does not extend another class

Software Engineering Observation 9.3

The Java compiler sets the superclass of a class to `Object` when the class declaration does not explicitly extend a superclass.

```
1  // Fig. 9.4: CommissionEmployee.java
2  // CommissionEmployee class represents a commission empl
3
4  public class CommissionEmployee extends Object
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percent
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs h
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate and store gross sales
21       setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first;
28    } // end method setFirstName
29
```

Declare `private` instance variables

Class `CommissionEmployee` `extends` class `Object`

Implicit call to `Object` constructor

Initialize instance variables

Invoke methods `setGrossSales` and `setCommissionRate` to validate data

Outline

issionEmployee .java

(1 of 4)

Line 4

Lines 6-10

Line 16

Lines 20-21

```java
   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

```

Outline

CommissionEmployee .java

(2 of 4)

```
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 commissionRate * grossSales;
88    } // end method earnings
89
```

Calculate earnings

Outline

CommissionEmployee.java

(3 of 4)

Lines 85-88

```
90    // return String representation of CommissionEmployee object
91    public String toString()
92    {
93       return String.format( "%s: %s %s\n%s: %s\n%s: %
94          "commission employee", firstName, lastName,
95          "social security number", socialSecurityNumber,
96          "gross sales", grossSales,
97          "commission rate", commissionRate );
98    } // end method toString
99 } // end class CommissionEmployee
```

Override method `toString` of class `Object`

Outline

`CommissionEmployee.java`

(4 of 4)

Lines 91-98

Common Programming Error 9.1

It is a syntax error to override a method with a more restricted access modifier—a `public` method of the superclass cannot become a `protected` or `private` method in the subclass; a `protected` method of the superclass cannot become a `private` method in the subclass. Doing so would break the "is-a" relationship in which it is required that all subclass objects be able to respond to method calls that are made to `public` methods declared in the superclass.(cont…)

Common Programming Error 9.1

If a `public` method could be overridden as a `protected` or `private` method, the subclass objects would not be able to respond to the same method calls as superclass objects. Once a method is declared `public` in a superclass, the method remains `public` for all that class’s direct and indirect subclasses.

◀ ▶

```
1  // Fig. 9.5: CommissionEmployeeTest.java
2  // Testing class CommissionEmployee.
3
4  public class CommissionEmployeeTest
5  {
6     public static void main( String args[] )
7     {
8        // instantiate CommissionEmployee object
9        CommissionEmployee employee = new CommissionEmployee(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // get commission employee data
13       System.out.println(
14          "Employee information obtained by get methods: \n" );
15       System.out.printf( "%s %s\n", "
16          employee.getFirstName() );
17       System.out.printf( "%s %s\n", "
18          employee.getLastName() );
19       System.out.printf( "%s %s\n", "Social security number is",
20          employee.getSocialSecurityNumber() );
21       System.out.printf( "%s %.2f\n", "Gross sales is"
22          employee.getGrossSales() );
23       System.out.printf( "%s %.2f\n", "Commiss
24          employee.getCommissionRate() );
25
26       employee.setGrossSales( 500 ); // set gross sales
27       employee.setCommissionRate( .1 ); // set commission rate
28
```

Instantiate `CommissionEmployee` object

Use `CommissionEmployee`'s *get* methods to retrieve the object's instance variable values

Use `CommissionEmployee`'s *set* methods to change the object's instance variable values

Outline

`CommissionEmployeeTest.java`

(1 of 2)

Lines 9-10

Lines 15-25

26-27

```
29       System.out.printf( "\n%s:\n\n%s\n",
30          "Updated employee information obtained by toString", employee );
31    } // end main
32 } // end class CommissionEmployeeTest
```

Implicitly call object's `toString` method

```
Employee information obtained by get methods:

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

Updated employee information obtained by toString:

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

Outline

onEmployee Test.java

(2 of 2)

Line 30

Program output

9.4.2 Creating a BasePlusCommissionEmployee Class without Using Inheritance

- Class BasePlusCommissionEmployee
  - Implicitly extends Object
  - Much of the code is similar to CommissionEmployee
    - private instance variables
    - public methods
    - constructor
  - Additions
    - private instance variable baseSalary
    - Methods setBaseSalary and getBaseSalary

Outline

```
 1  // Fig. 9.6: BasePlusCommissionEmployee.java
 2  // BasePlusCommissionEmployee class represents an employee that receives
 3  // a base salary in addition to commission.
 4
 5  public class BasePlusCommissionEmployee
 6  {
 7     private String firstName;
 8     private String lastName;
 9     private String socialSecurityNumber;
10     private double grossSales; // gross weekly sales
11     private double commissionRate; // commission percentage
12     private double baseSalary; // base salary per week
13
14     // six-argument constructor
15     public BasePlusCommissionEmployee( String first, String last,
16        String ssn, double sales, double rate, double salary )
17     {
18        // implicit call to Object constructor occurs here
19        firstName = first;
20        lastName = last;
21        socialSecurityNumber = ssn;
22        setGrossSales( sales ); // validate and store
23        setCommissionRate( rate ); // validate and store commission rate
24        setBaseSalary( salary ); // validate and store base salary
25     } // end six-argument BasePlusCommissionEmployee constructor
26
```

Add instance variable `baseSalary`

Use method `setBaseSalary` to validate data

`BasePlusCommission`
`Employee.java`

Line 12

Line 24

Outline

```
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

```

`BasePlusCommission Employee.java`

(2 of 4)

Outline

BasePlusCommissionEmployee.java

(3 of 4)

```java
   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

```

```
87    // set base salary
88    public void setBaseSalary( double salary )
89    {
90       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
91    } // end method setBaseSalary
92
93    // return base salary
94    public double getBaseSalary()
95    {
96       return baseSalary;
97    } // end method getBaseSalary
98
99    // calculate ea
100      public double
101      {
102         return baseSalary + ( commissionRate * grossSales );
103      } // end method earnings
104
105      // return String representation of BasePlusCommissionEmployee
106      public String toString()
107      {
108         return String.format(
109            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
110            "base-salaried commission employee", firstName, lastName,
111            "social security number", socialSecurityNumber,
112            "gross sales", grossSales, "commission rate'
113            "base salary", baseSalary );
114      } // end method toString
115   } // end class BasePlusCommissionEmployee
```

Method `setBaseSalary` validates data and sets instance variable `baseSalary`

Method `getBaseSalary` returns the value of instance variable `baseSalary`

Update method `earnings` to calculate the earnings of a base-salaried commission employee

Update method `toString` to display base salary

Outline

BasePlusCommissionEmployee.java

(4 of 4)

Lines 88-91

Lines 94-97

Line 102

Lines 108-113

Outline

```
 1  // Fig. 9.7: BasePlusCommissionEmployeeTest.java
 2  // Testing class BasePlusCommissionEmployee.
 3
 4  public class BasePlusCommissionEmployeeTest
 5  {
 6     public static void main( String args[] )
 7     {
 8        // instantiate BasePlusCommissionEmployee object
 9        BasePlusCommissionEmployee employee =
10           new BasePlusCommissionEmployee(
11           "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13        // get base-salaried commission employee data
14        System.out.println(
15           "Employee information obtained by get methods: \n" );
16        System.out.printf( "%s %s\n",
17           employee.getFirstName() );
18        System.out.printf( "%s %s\n",
19           employee.getLastName() );
20        System.out.printf( "%s %s\n", "Social security number is",
21           employee.getSocialSecurityNumber() );
22        System.out.printf( "%s %.2f\n", "Gross sales is",
23           employee.getGrossSales() );
24        System.out.printf( "%s %.2f\n", "Commission rate is",
25           employee.getCommissionRate() );
26        System.out.printf( "%s %.2f\n", "Base salary is",
27           employee.getBaseSalary() );
28
```

Instantiate `BasePlusCommissionEmployee` object

Use `BasePluCommissionEmployee`’s *get* methods to retrieve the object’s instance variable values

sePlusCommission
ployeeTest.java

(1 of 2)

Line 9-11

Lines 16-27

Outline

```
29       employee.setBaseSalary( 1000 ); // set base salary
30
31       System.out.printf( "\n%s:\n\n%s\n",
32          "Updated employee information obta
33           employee.toString() );
34    } // end main
35 } // end class BasePlusCommissionEmplo
```

```
Employee information obtained by get met

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

Updated employee information obtained by toString:

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

Use `BasePlusCommissionEmployee`’s `setBaseSalary` methods to set base salary

Explicitly call object’s `toString` method

`BasePlusCommissionEmployeeTest.java`

(2 of 2)

Line 29

Line 33

Program output

Software Engineering Observation 9.4

Copying and pasting code from one class to another can spread errors across multiple source code files. To avoid duplicating code (and possibly errors), use inheritance, rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the instance variables and methods of another class.

Software Engineering Observation 9.5

With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are required for these common features, software developers need only to make the changes in the superclass—subclasses then inherit the changes. Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

9.4.3 Creating a CommissionEmployee-BasePlusCommiionEmployee Inheritance Hierarchy

- Class BasePlusCommissionEmployee2
  - Extends class CommissionEmployee
  - Is a CommissionEmployee
  - Has instance variable baseSalary
  - Inherits public and protected members
  - Constructor not inherited

Outline

```
1  // Fig. 9.8: BasePlusCommissionEmployee2.java
2  // BasePlusCommissionEmployee2 inherits from class CommissionEmployee.
3
4  public class BasePlusCommissionEmployee2 extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee2( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       // explicit call to superclass CommissionEmployee constructor
13       super( first, last, ssn, sales, rate );
14
15       setBaseSalary( amount ); // validate and store base salary
16    } // end six-argument BasePlusCommissi
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
```

Class `BasePluCommissionEmployee2` is a subclass of `CommissionEmployee`

Invoke the superclass constructor using the superclass constructor call syntax

`BasePlusCommission`
`Employee2.java`

Line 4

Line 13

Outline

```
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       // not allowed: commissionRate and grossSales private in superclass
34       return baseSalary + ( commissionRate * grossSales );
35    } // end method earnings
36
37    // return String representation
38    public String toString()
39    {
40       // not allowed: attempts to a
41       return String.format(
42          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
43          "base-salaried commission employee", firstName, lastName,
44          "social security number", socialSecurityNumber,
45          "gross sales", grossSales, "commission rate", commissionRate,
46          "base salary", baseSalary );
47    } // end method toString
48 } // end class BasePlusCommissionEmployee2
```

Compiler generates errors because superclass's instance variable `commissionRate` and `grossSales` are `private`

Compiler generates errors because superclass's instance variable `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` are `private`

BasePlusCommission

(2 of 3)

Line 34

Lines 41-46

```
BasePlusCommissionEmployee2.java:34: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee2.java:34: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee2.java:43: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee2.java:43: lastName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee2.java:44: socialSecurityNumber has private access in
CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee2.java:45: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee2.java:45: commissionRate has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                       ^
7 errors
```

Outline

`BasePlusCommissionEmployee2.java`

(3 of 3)

Compiler generated errorss

Common Programming Error 9.2

A compilation error occurs if a subclass constructor calls one of its superclass constructors with arguments that do not match exactly the number and types of parameters specified in one of the superclass constructor declarations.

9.4.4 CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables

- **Use `protected` instance variables**
  - **Enable class `BasePlusCommissionEmployee` to directly access superclass instance variables**
  - **Superclass's `protected` members are inherited by all subclases of that superclass**

```
1  // Fig. 9.9: CommissionEmployee2.java
2  // CommissionEmployee2 class represents a commission employee.
3
4  public class CommissionEmployee2
5  {
6     protected String firstName;
7     protected String lastName;
8     protected String socialSecurityNumber;
9     protected double grossSales; // gross weekly sales
10    protected double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee2( 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 CommissionEmployee2 constructor
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first;
28    } // end method setFirstName
29
```

Declare `protected` instance variables

Outline

`CommissionEmployee2.java`

(1 of 4)

Line 6-10

```
   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

```

Outline

Commission

Employee2.java

(2 of 4)

Outline

```java
   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
      return commissionRate * grossSales;
   } // end method earnings

```

Commission

Employee2.java

(3 of 4)

Outline

```
   // return String representation of CommissionEmployee2 object
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
         "commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales,
         "commission rate", commissionRate );
   } // end method toString
} // end class CommissionEmployee2
```

Commission

Employee2.java

(4 of 4)

Outline

```
1  // Fig. 9.10: BasePlusCommissionEmployee3.java
2  // BasePlusCommissionEmployee3 inherits from CommissionEmployee2 and has
3  // access to CommissionEmployee2's protected members.
4
5  public class BasePlusCommissionEmployee3 extends CommissionEmployee2
6  {
7     private double baseSalary; // base salary per week
8
9     // six-argument constructor
10    public BasePlusCommissionEmployee3( String first, Strin
11       String ssn, double sales, double rate, double salary
12    {
13       super( first, last, ssn, sales, rate );
14       setBaseSalary( salary ); // validate and store base salary
15    } // end six-argument BasePlusCommissionEmployee3 constructor
16
17    // set base salary
18    public void setBaseSalary( double salary )
19    {
20       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21    } // end method setBaseSalary
22
23    // return base salary
24    public double getBaseSalary()
25    {
26       return baseSalary;
27    } // end method getBaseSalary
28
```

Must call superclass's constructor

BasePlusCommissionEmployee3.java

of 2)

Line 13

```
29    // calculate earnings
30    public double earnings()
31    {
32       return baseSalary + ( commissionRate * grossSales );
33    } // end method earnings
34
35    // return String representation of BasePlusCommissionEmployee3
36    public String toString()
37    {
38       return String.format(
39          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
40          "base-salaried commission employee", firstName, lastName,
41          "social security number", socialSecurityNumber,
42          "gross sales", grossSales, "commission rate", commissionRate,
43          "base salary", baseSalary );
44    } // end method toString
45 } // end class BasePlusCommissionEmployee3
```

Directly access superclass's `protected` instance variables

Outline

`BasePlusCommissionEmployee3.java`

Line 32

Lines 38-43

◀ ▶

```
// Fig. 9.11: BasePlusCommissionEmployeeTest3.java
// Testing class BasePlusCommissionEmployee3.

public class BasePlusCommissionEmployeeTest3
{
   public static void main( String args[] )
   {
      // instantiate BasePlusCommissionEmployee3 object
      BasePlusCommissionEmployee3 employee =
         new BasePlusCommissionEmployee3(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // get base-salaried commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );
      System.out.printf( "%s %.2f\n", "Base salary is",
         employee.getBaseSalary() );

```

Outline

BasePlusCommission EmployeeTest3.java

(1 of 2)

Outline

```java
      employee.setBaseSalary( 1000 ); // set base salary

      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained by toString",
         employee.toString() );
   } // end main
} // end class BasePlusCommissionEmployeeTest3
```

```
Employee information obtained by get methods:

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

Updated employee information obtained by toString:

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

BasePlusCommission EmployeeTest3.java

(2 of 2)

Program output

9.4.4 `CommissionEmployee-BasePlusCommissionEmployee` Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Using `protected` instance variables
  - Advantages
    - subclasses can modify values directly
    - Slight increase in performance
      - Avoid set/get method call overhead
  - Disadvantages
    - No validity checking
      - subclass can assign illegal value
    - Implementation dependent
      - subclass methods more likely dependent on superclass implementation
      - superclass implementation changes may result in subclass modifications
        - Fragile (brittle) software

Software Engineering Observation 9.6

Use the `protected` access modifier when a superclass should provide a method only to its subclasses and other classes in the same package, but not to other clients.

Software Engineering Observation 9.7

Declaring superclass instance variables `private` (as opposed to `protected`) enables the superclass implementation of these instance variables to change without affecting subclass implementations.

Error-Prevention Tip 9.1

When possible, do not include `protected` instance variables in a superclass. Instead, include non-`private` methods that access `private` instance variables. This will ensure that objects of the class maintain consistent states.

9.4.5 CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Uing private Instance Variables

- **Reexamine hierarchy**
  - **Use the best software engineering practice**
    - **Declare instance variables as `private`**
    - **Provide public *get* and *set* methods**
    - **Use *get* method to obtain values of instance variables**

```java
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

Outline

`Commission`

`Employee3.java`

(1 of 4)

Lines 6-10

```
   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

```

Commission

Employee3.java

(2 of 4)

Outline

```
   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

```

`Commission`

`Employee3.java`

(3 of 4)

```
84    // calculate earnings
85    public double earnings()
86    {
87       return getCommissionRate() * getGrossSales();
88    } // end method earnings
89
90    // return String representation of CommissionEmployee
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
```

Use *get* methods to obtain the values of instance variables

Outline

ission

Employee3.java

(4 of 4)

Line 87

Lines 94-97

◀ ▶

Outline

```java
// Fig. 9.13: BasePlusCommissionEmployee4.java
// BasePlusCommissionEmployee4 class inherits from CommissionEmployee3 and
// accesses CommissionEmployee3's private data via CommissionEmployee3's
// public methods.

public class BasePlusCommissionEmployee4 extends CommissionEmployee3
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee4( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee4 constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

```

Inherits from `CommissionEmployee3`

BasePlusCommissionEmployee4.java

)

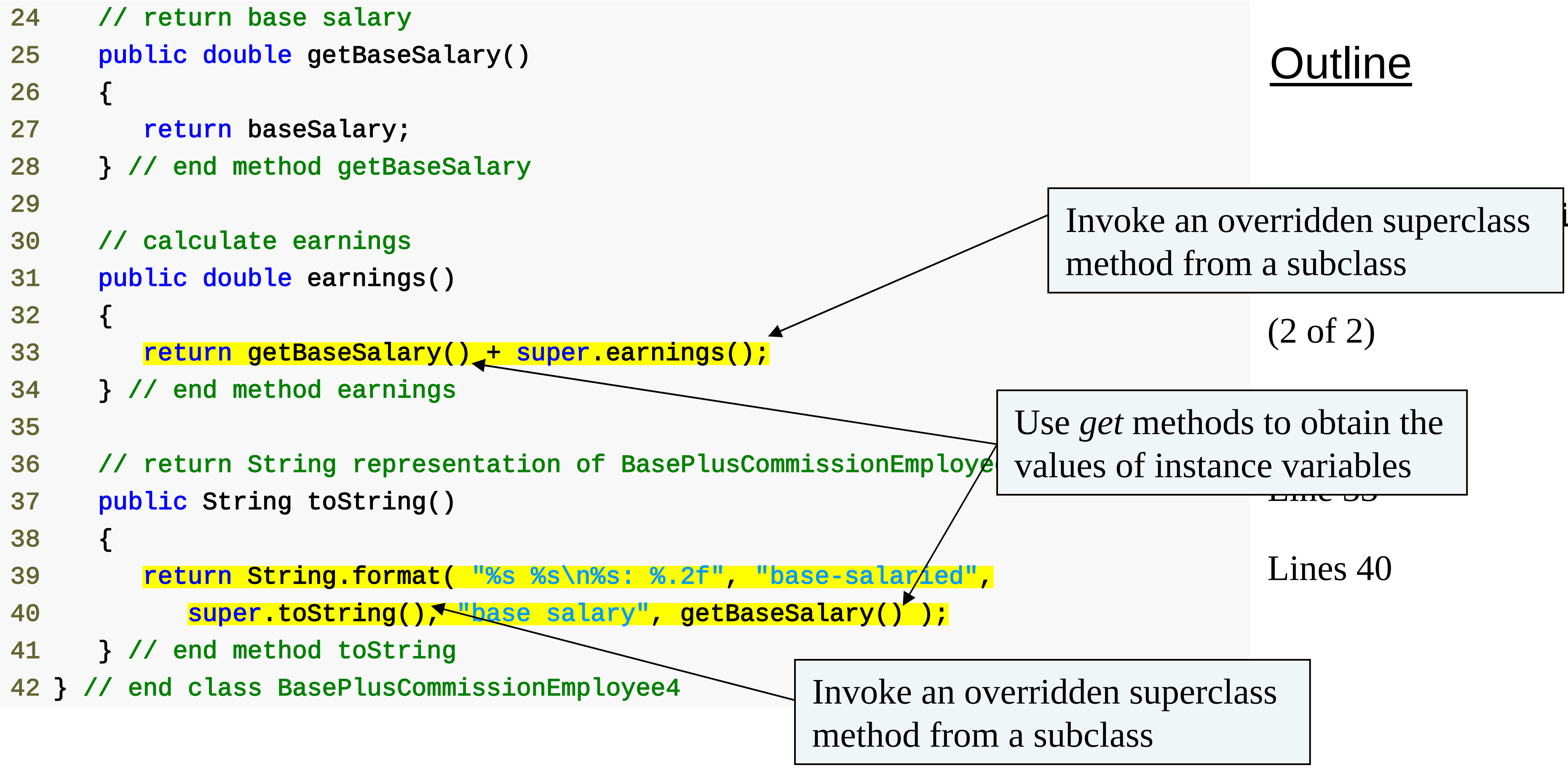
Outline

ion

(2 of 2)

Line 33

Lines 40



```
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 BasePlusCommissionEmploye
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
```

Invoke an overridden superclass method from a subclass

Use *get* methods to obtain the values of instance variables

Invoke an overridden superclass method from a subclass

Common Programming Error 9.3

When a superclass method is overridden in a subclass, the subclass version often calls the superclass version to do a portion of the work. Failure to prefix the superclass method name with the keyword `super` and a dot (`.`) separator when referencing the superclass's method causes the subclass method to call itself, creating an error called infinite recursion. Recursion, used correctly, is a powerful capability discussed in Chapter 15, Recursion.

Outline

ssion
.java

```java
1  // Fig. 9.14: BasePlusCommissionEmployeeTest4.java
2  // Testing class BasePlusCommissionEmployee4.
3
4  public class BasePlusCommissionEmployeeTest4
5  {
6     public static void main( String args[] )
7     {
8        // instantiate BasePlusCommissionEmployee4 object
9        BasePlusCommissionEmployee4 employee =
10          new BasePlusCommissionEmployee4(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // get base-salaried commission employee data
14       System.out.println(
15          "Employee information obtained by get methods: \n" );
16       System.out.printf( "%s %s\n", "First name is",
17          employee.getFirstName() );
18       System.out.printf( "%s %s\n", "Last name is",
19          employee.getLastName() );
20       System.out.printf( "%s %s\n", "Social security number is",
21          employee.getSocialSecurityNumber() );
22       System.out.printf( "%s %.2f\n", "Gross sales is",
23          employee.getGrossSales() );
24       System.out.printf( "%s %.2f\n", "Commission rate is",
25          employee.getCommissionRate() );
26       System.out.printf( "%s %.2f\n", "Base salary
27          employee.getBaseSalary() );
28
```

Create `BasePlusCommissionEmployee4` object.

Lines 9-11

Lines 16-25

Use inherited *get* methods to access inherited `private` instance variables

Use `BasePlusCommissionEmployee4` *get* method to access `private` instance variable.

```
      employee.setBaseSalary( 1000 ); // set base salary

      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained
         employee.toString() );
   } // end main
} // end class BasePlusCommissionEmployeeTest4
```

```
Employee information obtained by get methods:

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

Updated employee information obtained by toString:

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

Use `BasePlusCommissionEmployee4` *set* method to modify `private` instance variable `baseSalary`.

Outline

on
EmployeeTest4.java

(2 of 2)

9.5 Constructors in Subclasses

- Instantiating subclass object
  - Chain of constructor calls
    - subclass constructor invokes superclass constructor
      - Implicitly or explicitly
    - Base of inheritance hierarchy
      - Last constructor called in chain is `Object`'s constructor
      - Original subclass constructor's body finishes executing last
      - Example: `CommissionEmployee3`-`BasePlusCommissionEmployee4` hierarchy
        - `CommissionEmployee3` constructor called second last (last is `Object` constructor)
        - `CommissionEmployee3` constructor's body finishes execution second (first is `Object` constructor's body)

Software Engineering Observation 9.8

When a program creates a subclass object, the subclass constructor immediately calls the superclass constructor (explicitly, via `super`, or implicitly). The superclass constructor’s body executes to initialize the superclass’s instance variables that are part of the subclass object, then the subclass constructor’s body executes to initialize the subclass-only instance variables.(cont…)

Software Engineering Observation 9.8

Java ensures that even if a constructor does not assign a value to an instance variable, the variable is still initialized to its default value (e.g., `0` for primitive numeric types, `false` for `booleans`, `null` for references).

```
1  // Fig. 9.15: CommissionEmployee4.java
2  // CommissionEmployee4 class represents a commission employee.
3
4  public class CommissionEmployee4
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 CommissionEmployee4( 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 ar
21       setCommissionRate( rate ); // validate
22
23       System.out.printf(
24          "\nCommissionEmployee4 constructor:\n%s\n", this );
25    } // end five-argument CommissionEmployee4 constructor
26
```

Constructor outputs message to demonstrate method call order.

Outline

`CommissionEmployee4.java`

(1 of 4)

Lines 23-24

Outline

```
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

```

`CommissionEmployee4.java`

(2 of 4)

```java
   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

```

Outline

`CommissionEmployee4.java`

(3 of 4)

```java
   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

   // return String representation of CommissionEmployee4 object
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
         "commission employee", getFirstName(), getLastName(),
         "social security number", getSocialSecurityNumber(),
         "gross sales", getGrossSales(),
         "commission rate", getCommissionRate() );
   } // end method toString
} // end class CommissionEmployee4
```

Outline

CommissionEmployee4.java

(4 of 4)

Outline

```
1  // Fig. 9.16: BasePlusCommissionEmployee5.java
2  // BasePlusCommissionEmployee5 class declaration.
3
4  public class BasePlusCommissionEmployee5 extends CommissionEmployee4
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee5( 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 a
14
15       System.out.printf(
16          "\nBasePlusCommissionEmployee5 constructor:\n%s\n", this );
17    } // end six-argument BasePlusCommissionEmployee5 constructor
18
19    // set base salary
20    public void setBaseSalary( double salary )
21    {
22       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23    } // end method setBaseSalary
24
```

Constructor outputs message to demonstrate method call order.

BasePlusCommissionEmployee5.java

(1 of 2)

Lines 15-16

```java
   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee5
   public String toString()
   {
      return String.format( "%s %s\n%s: %.2f", "base-salaried",
         super.toString(), "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee5
```

Outline

`BasePlusCommission`
`Employee5.java`

(2 of 2)

Outline

```java
// Fig. 9.17: ConstructorTest.java
// Display order in which superclass and subclass constructors are called.

public class ConstructorTest
{
   public static void main( String args[] )
   {
      CommissionEmployee4 employee1 = new CommissionEmployee4(
         "Bob", "Lewis", "333-33-3333", 5000, .04 );

      System.out.println();
      BasePlusCommissionEmployee5 employee2 =
         new BasePlusCommissionEmployee5(
         "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );

      System.out.println();
      BasePlusCommissionEmployee5 employee3 =
         new BasePlusCommissionEmployee5(
         "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
   } // end main
} // end class ConstructorTest
```

Instantiate `CommissionEmployee4` object

Instantiate two `BasePlusCommissionEmployee5` objects to demonstrate order of subclass and superclass constructor method calls.

`ConstructorTest`

`.java`

(1 of 2)

Outline

ConstructorTest

.java

(2 of 2)

```
CommissionEmployee4 constructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee4 constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 0.00

BasePlusCommissionEmployee5 constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00


CommissionEmployee4 constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 0.00

BasePlusCommissionEmployee5 constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00
```

Subclass `BasePlusCommissionEmployee5` constructor body executes after superclass `CommissionEmployee4`’s constructor finishes execution.

9.6 Software Engineering with Inheritance

- Customizing existing software
  - Inherit from existing classes
    - Include additional members
    - Redefine superclass members
    - No direct access to superclass’s source code
      - Link to object code
  - Independent software vendors (ISVs)
    - Develop proprietary code for sale/license
      - Available in object-code format
    - Users derive new classes
      - Without accessing ISV proprietary source code

Software Engineering Observation 9.9

Despite the fact that inheriting from a class does not require access to the class's source code, developers often insist on seeing the source code to understand how the class is implemented. Developers in industry want to ensure that they are extending a solid class—for example, a class that performs well and is implemented securely.

Software Engineering Observation 9.10

At the design stage in an object-oriented system, the designer often finds that certain classes are closely related. The designer should “factor out” common instance variables and methods and place them in a superclass. Then the designer should use inheritance to develop subclasses, specializing them with capabilities beyond those inherited from the superclass.

Software Engineering Observation 9.11

Declaring a subclass does not affect its superclass's source code. Inheritance preserves the integrity of the superclass.

Software Engineering Observation 9.12

Just as designers of non-object-oriented systems should avoid method proliferation, designers of object-oriented systems should avoid class proliferation. Such proliferation creates management problems and can hinder software reusability, because in a huge class library it becomes difficult for a client to locate the most appropriate classes. The alternative is to create fewer classes that provide more substantial functionality, but such classes might prove cumbersome.

Performance Tip 9.1

If subclasses are larger than they need to be (i.e., contain too much functionality), memory and processing resources might be wasted. Extend the superclass that contains the functionality that is closest to what is needed.

◀ ▶

9.7 Object Class

- Class Object methods
  - clone
  - equals
  - finalize
  - getClass
  - hashCode
  - notify, notifyAll, wait
  - toString

Method	Description
`Clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it is called. When cloning is required for objects of a class, the class should override method `clone` as a `public` method and should implement interface `Cloneable` (package `java.lang`). The default implementation of this method performs a so-called shallow copy—instance variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a deep copy that creates a new object for each reference type instance variable. There are many subtleties to overriding method `clone`. You can learn more about cloning in the following article: `java.sun.com/developer/JDCTechTips/2001/tt0306.html`

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 1 of 4)

Method	Description
Equals	This method compares two objects for equality and returns true if they are equal and false otherwise. The method takes any Object as an argument. When objects of a particular class must be compared for equality, the class should override method equals to compare the contents of the two objects. The method's implementation should meet the following requirements: • It should return false if the argument is null. • It should return true if an object is compared to itself, as in object1.equals(object1). • It should return true only if both object1.equals(object2) and object2.equals(object1) would return true. • For three objects, if object1.equals(object2) returns true and object2.equals(object3) returns true, then object1.equals(object3) should also return true. • If equals is called multiple times with the two objects and the objects do not change, the method should consistently return true if the objects are equal and false otherwise. A class that overrides equals should also override hashCode to ensure that equal objects have identical hashcodes. The default equals implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 29.3.3 demonstrates class String's equals method and differentiates between comparing String objects with == and with equals.

Fig. 9.18 | Object methods that are inherited directly or indirectly by all classes. (Part 2 of 4)

Method	Description
`finalize`	This `protected` method (introduced in Section 8.10 and Section 8.11) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object’s memory. It is not guaranteed that the garbage collector will reclaim an object, so it cannot be guaranteed that the object’s `finalize` method will execute. The method must specify an empty parameter list and must return `void`. The default implementation of this method serves as a placeholder that does nothing.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Section 10.5 and Section 21.3) returns an object of class `Class` (package `java.lang`) that contains information about the object’s type, such as its class name (returned by `Class` method `getName`). You can learn more about class `Class` in the online API documentation at `java.sun.com/j2se/5.0/docs/api/java/lang/Class.html`.

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 3 of 4)

Method	Description
`hashCode`	A hashtable is a data structure (discussed in Section 19.10) that relates one object, called the key, to another object, called the value. When initially inserting a value into a hashtable, the key's `hashCode` method is called. The hashcode value returned is used by the hashtable to determine the location at which to insert the corresponding value. The key's hashcode is also used by the hashtable to locate the key's corresponding value.
`notify`, `notifyAll`, `wait`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23. In J2SE 5.0, the multithreading model has changed substantially, but these features continue to be supported.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 4 of 4)

9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels

- Labels
  - Display information and instructions
  - `JLabel`
    - Display a single line of text
    - Display an image
    - Display both text and image

Outline

`LabelDemo.java`

(1 of 2)

```
// Fig 9.19: LabelDemo.java
// Demonstrates the use of labels.
import java.awt.BorderLayout;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JFrame;

public class LabelDemo
{
   public static void main( String args[] )
   {
      // Create a label with plain text
      JLabel northLabel = new JLabel( "North" );

      // create an icon from an image so we can put it on a JLabel
      ImageIcon labelIcon = new ImageIcon( "GUItip.gif" );

      // create a label with an Icon instead of text
      JLabel centerLabel = new JLabel( labelIcon );

      // create another label with an Icon
      JLabel southLabel = new JLabel( labelIcon

      // set the label to display text (as well as an icon)
      southLabel.setText( "South" );

```

Create a `JLabel` that displays the string “North”

`ImageIcon` constructor argument specifies the path to the image

Declare and initialize `centerLabel` with a `JLabel` that displays the `labelIcon`

Change the text the `southLabel` displays

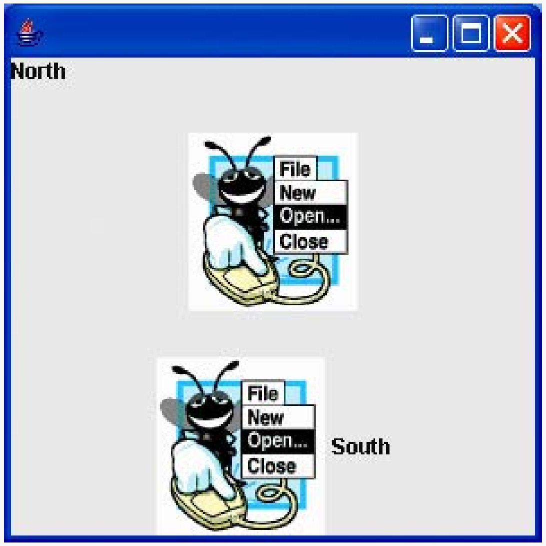
```
27       // create a frame to hold the labels
28       JFrame application = new JFrame();
29
30       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
31
32       // add the labels to the frame; the second argument specifies
33       // where on the frame to add the label
34       application.add( northLabel, BorderLayout.NORTH );
35       application.add( centerLabel, BorderLayout.CENTER );
36       application.add( southLabel, BorderLayout.SOUTH );
37
38       application.setSize( 300, 300 ); // set the size of the frame
39       application.setVisible( true ); // show the frame
40    } // end main
41 } // end class LabelDemo
```

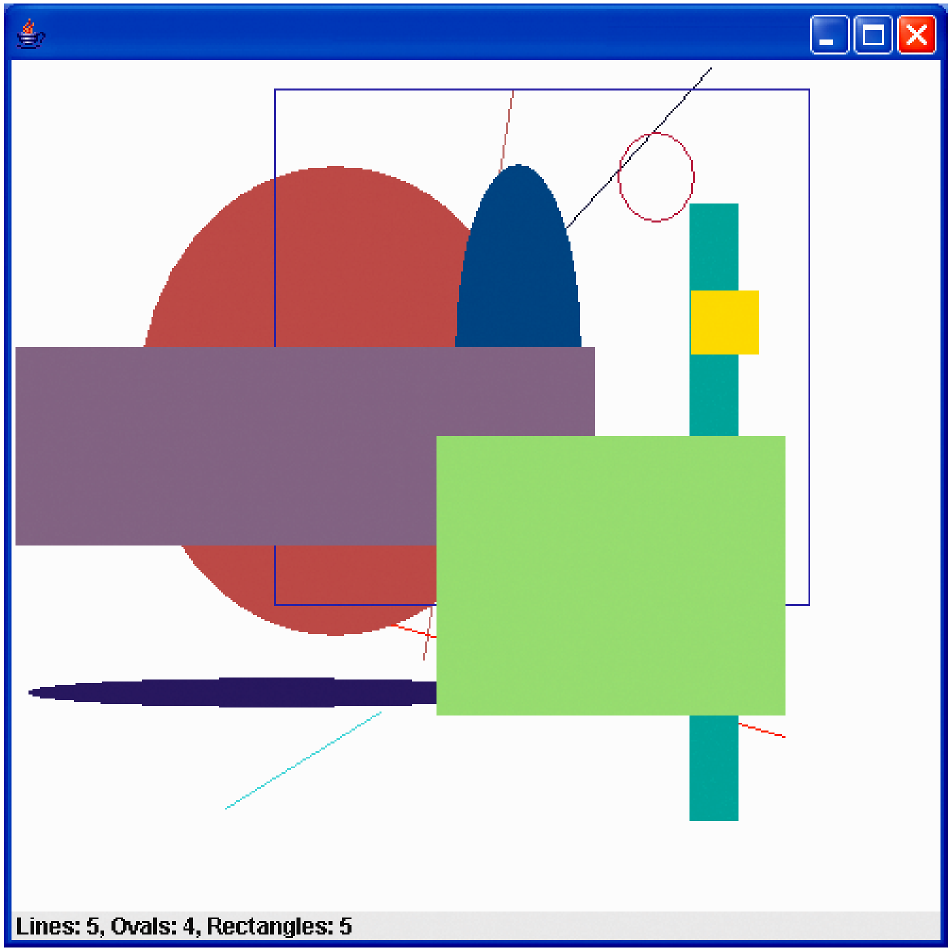
Outline

Attach the labels to the `JFrame` at north, center and south

(2 of 2)

Lines 34-36





Lines: 5, Ovals: 4, Rectangles: 5

Fig. 9.20 | `JLabel` displaying shape statistics.