Object-Oriented Programming: Inheritance
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.
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
**Fig. 9.1 | More Inheritance examples.**

<table>
<thead>
<tr>
<th>Superclass</th>
<th>Subclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>GraduateStudent, UndergraduateStudent</td>
</tr>
<tr>
<td>Shape</td>
<td>Circle, Triangle, Rectangle</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan, HomeImprovementLoan, MortgageLoan</td>
</tr>
<tr>
<td>Employee</td>
<td>Faculty, Staff</td>
</tr>
<tr>
<td>BankAccount</td>
<td>CheckingAccount, SavingsAccount</td>
</tr>
</tbody>
</table>
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 Community Members
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.
// Fig. 9.4: CommissionEmployee.java
// CommissionEmployee class represents a commission employee.

public class CommissionEmployee extends Object 
{
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;
    private double grossSales; // gross weekly sales
    private double commissionRate; // commission percent

    // five-argument constructor
    public CommissionEmployee( String first, String last, String ssn,
                               double sales, double rate ) 
    
    { // implicit call to Object constructor occurs here
        firstName = first;
        lastName = last;
        socialSecurityNumber = ssn;
        setGrossSales( sales ); // validate and store gross sales
        setCommissionRate( rate ); // validate and store commission rate
    } // end five-argument CommissionEmployee constructor

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

} // end class CommissionEmployee
```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
```
// 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
90    // return String representation of CommissionEmployee object
91    public String toString()
92    {
93       return String.format( "%s: %s %s
%s: %s
%s: %.2f
%s: %.2f",
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
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.
public class CommissionEmployeeTest {
    public static void main( String args[] ) {
        // instantiate CommissionEmployee object
        CommissionEmployee employee = new CommissionEmployee(
            "Sue", "Jones", "222-22-2222", 10000, .06 );

        // get 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() );

        employee.setGrossSales( 500 ); // set gross sales
        employee.setCommissionRate( .1 ); // set commission rate
    }
}
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
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
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     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     private double baseSalary;  // base salary per week
12
13     // six-argument constructor
14     public BasePlusCommissionEmployee( String first, String last, 
15         String ssn, double sales, double rate, double salary )
16     {
17         // implicit call to Object constructor occurs here
18         firstName = first;
19         lastName = last;
20         socialSecurityNumber = ssn;
21         setGrossSales( sales );  // validate and store gross sales
22         setCommissionRate( rate );  // validate and store commission rate
23         setBaseSalary( salary );  // validate and store base salary
24     } // end six-argument BasePlusCommissionEmployee constructor
public void setFirstName(String first) {
    firstName = first;
} // end method setFirstName

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

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

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

public void setSocialSecurityNumber(String ssn) {
    socialSecurityNumber = ssn; // should validate
} // end method setSocialSecurityNumber
// 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
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.
// Fig. 9.7: BasePlusCommissionEmployeeTest.java
// Testing class BasePlusCommissionEmployee.

public class BasePlusCommissionEmployeeTest {
    public static void main( String args[] ) {
        // instantiate BasePlusCommissionEmployee object
        BasePlusCommissionEmployee employee =
            new BasePlusCommissionEmployee(
                "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() );
    }
}
employee.setBaseSalary( 1000 ); // set base salary

System.out.printf("%s
%s
", "Updated employee information obtained by toString", employee.toString());

} // end main

} // end class BasePlusCommissionEmployeeTest

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
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-BasePlusCommissionEmployee` Inheritance Hierarchy

- Class `BasePlusCommissionEmployee2`
  - Extends class `CommissionEmployee`
  - Is a `CommissionEmployee`
  - Has instance variable `baseSalary`
  - Inherits public and protected members
  - Constructor not inherited
```java
public class BasePlusCommissionEmployee2 extends CommissionEmployee {
    private double baseSalary; // base salary per week

    // six-argument constructor
    public BasePlusCommissionEmployee2(String first, String last, String ssn, double sales, double rate, double salary) {
        super(first, last, ssn, sales, rate);
        setBaseSalary(amount); // validate and store base salary
    } // end six-argument BasePlusCommissionEmployee2 constructor

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

Class **BasePlusCommissionEmployee2** is a subclass of **CommissionEmployee**. To invoke the superclass constructor, use the superclass constructor call syntax. For example, in the six-argument constructor, `super(first, last, ssn, sales, rate);` invokes the constructor of **CommissionEmployee**. The `setBaseSalary(double salary)` method sets the base salary, ensuring it's not negative.
// return base salary
public double getBaseSalary()
{
    return baseSalary;
} // end method getBaseSalary

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

// return String representation
public String toString()
{
    return String.format("%s: %s %s
%s: %s
%s: %.2f
%s: %.2f
%s: %.2f",
    "base-salaried commission employee", firstName, lastName,
    "social security number", socialSecurityNumber,
    "gross sales", grossSales, "commission rate", commissionRate,
    "base salary", baseSalary);
} // end method toString

// end class BasePlusCommissionEmployee2
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
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
subclasses of that superclass
// Fig. 9.9: CommissionEmployee2.java
// CommissionEmployee2 class represents a commission employee.

public class CommissionEmployee2
{
    protected String firstName;
    protected String lastName;
    protected String socialSecurityNumber;
    protected double grossSales; // gross weekly sales
    protected double commissionRate; // commission percentage

    // five-argument constructor
    public CommissionEmployee2( String first, String last, String ssn,
                                 double sales, double rate )
    {
        // implicit call to Object constructor occurs here
        firstName = first;
        lastName = last;
        socialSecurityNumber = ssn;
        setGrossSales( sales ); // validate and store gross sales
        setCommissionRate( rate ); // validate and store commission rate
    } // end five-argument CommissionEmployee2 constructor

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

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

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

    // set gross sales
    public void setGrossSales( double sales )
    {
        grossSales = sales;
    } // end method setGrossSales

    // set commission rate
    public void setCommissionRate( double rate )
    {
        commissionRate = rate;
    } // end method setCommissionRate

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

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

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

    // get gross sales
    public double getGrossSales() { return grossSales; }

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

    // to string
    public String toString()
    {
        return "First Name: " + firstName + " Last Name: " + lastName + " SSN: " + socialSecurityNumber + " Gross Sales: "+ grossSales + " Commission Rate: " + commissionRate;
    } // end method toString

    // test CommissionEmployee2 class
    public static void main( String[] args )
    {
        CommissionEmployee2 ce = new CommissionEmployee2( "John", "Doe", "111-22-3333", 1000.0, 0.05 );
        System.out.println( ce );
    } // end main
} // end class CommissionEmployee2

// Figure 9.9: CommissionEmployee2.java
// CommissionEmployee2 class represents a commission employee.

// five-argument constructor
public CommissionEmployee2( String first, String last, String ssn, double sales, double rate )
{
    // implicit call to Object constructor occurs here
    firstName = first;
    lastName = last;
    socialSecurityNumber = ssn;
    setGrossSales( sales ); // validate and store gross sales
    setCommissionRate( rate ); // validate and store commission rate
} // end five-argument CommissionEmployee2 constructor

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

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

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

// set gross sales
public void setGrossSales( double sales )
{
    grossSales = sales;
} // end method setGrossSales

// set commission rate
public void setCommissionRate( double rate )
{
    commissionRate = rate;
} // end method setCommissionRate

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

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

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

// get gross sales
public double getGrossSales() { return grossSales; }

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

// to string
public String toString()
{
    return "First Name: " + firstName + " Last Name: " + lastName + " SSN: " + socialSecurityNumber + " Gross Sales: "+ grossSales + " Commission Rate: " + commissionRate;
} // end method toString

// test CommissionEmployee2 class
public static void main( String[] args )
{
    CommissionEmployee2 ce = new CommissionEmployee2( "John", "Doe", "111-22-3333", 1000.0, 0.05 );
    System.out.println( ce );
} // end main

30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // should validate
52    } // end method setSocialSecurityNumber
53
54    // return social security number
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // end method getSocialSecurityNumber
59
60  // set gross sales amount
61  public void setGrossSales( double sales )
62  {
63     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64  } // end method setGrossSales
65
66  // return gross sales amount
67  public double getGrossSales()
68  {
69     return grossSales;
70  } // end method getGrossSales
71
72  // set commission rate
73  public void setCommissionRate( double rate )
74  {
75     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76  } // end method setCommissionRate
77
78  // return commission rate
79  public double getCommissionRate()
80  {
81     return commissionRate;
82  } // end method getCommissionRate
83
84  // calculate earnings
85  public double earnings()
86  {
87     return commissionRate * grossSales;
88  } // end method earnings
90    // return String representation of CommissionEmployee2 object
91    public String toString()
92    {
93       return String.format( "%s: %s %s
%s: %s
%s: %.2f
%s: %.2f",
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 CommissionEmployee2
public class BasePlusCommissionEmployee3 extends CommissionEmployee2 {
    private double baseSalary; // base salary per week

    // six-argument constructor
    public BasePlusCommissionEmployee3(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 BasePlusCommissionEmployee3 constructor

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

    // return base salary
    public double getBaseSalary()
    {
        return baseSalary;
    } // end method getBaseSalary
// calculate earnings
public double earnings()
{
    return baseSalary + (commissionRate * grossSales);
} // end method earnings

// return String representation of BasePlusCommissionEmployee3
public String toString()
{
    return String.format("%s: %s %s
%s: %s
%s: %.2f
%s: %.2f
%s: %.2f",
"base-salaried commission employee", firstName, lastName,
"social security number", socialSecurityNumber,
"gross sales", grossSales, "commission rate", commissionRate,
"base salary", baseSalary);
} // end method toString

} // end 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());
    }
}
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
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.
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

Using 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
// Fig. 9.12: CommissionEmployee3.java
// CommissionEmployee3 class represents a commission employee.

public class CommissionEmployee3
{
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;
    private double grossSales; // gross weekly sales
    private double commissionRate; // commission percentage

    // five-argument constructor
    public CommissionEmployee3( String first, String last, String ssn,
                                 double sales, double rate )
    {
        // implicit call to Object constructor occurs here
        firstName = first;
        lastName = last;
        socialSecurityNumber = ssn;
        setGrossSales( sales ); // validate and store gross sales
        setCommissionRate( rate ); // validate and store commission rate
    } // end five-argument CommissionEmployee3 constructor

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

    // Declare private instance variables
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // should validate
52    } // end method setSocialSecurityNumber
53
54    // return social security number
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // end method getSocialSecurityNumber
59
Commission

Employee3.java

(3 of 4)

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

// return String representation of CommissionEmployee3 object
public String toString()
{
    return String.format( "%s: %s %s
%s: %s
%s: %.2f
%s: %.2f",
    "commission employee", getFirstName(), getLastName(),
    "social security number", getSocialSecurityNumber(),
    "gross sales", getGrossSales(),
    "commission rate", getCommissionRate() );
} // end method toString

} // end class CommissionEmployee3
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
// 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 BasePlusCommissionEmployee
public String toString()
{
    return String.format( "%s %s
%s: %.2f", "base-salaried", super.toString(), "base salary", getBaseSalary() );
} // end method toString

// end class BasePlusCommissionEmployee4
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.
public class BasePlusCommissionEmployeeTest4 {
    public static void main( String args[] ) {
        // instantiate BasePlusCommissionEmployee4 object
        BasePlusCommissionEmployee4 employee = new BasePlusCommissionEmployee4("Bob", "Lewis", "333-33-3333", 5000, .04, 300);

        // get base-salaried commission employee data
        System.out.println("Employee information obtained by get methods: 
");
        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());
    }
}
29   employee.setBaseSalary(1000); // set base salary
30
31   System.out.printf("\n%s:  \\
32         "Updated employee information obtained by toString
33               employee.toString());
34 } // end main
35 } // 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.
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)
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).
public class CommissionEmployee4
{
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;
    private double grossSales; // gross weekly sales
    private double commissionRate; // commission percentage

    // five-argument constructor
    public CommissionEmployee4( String first, String last, String ssn,
        double sales, double rate )
    {
        // implicit call to Object constructor occurs here
        firstName = first;
        lastName = last;
        socialSecurityNumber = ssn;
        setGrossSales( sales ); // validate and store gross sales
        setCommissionRate( rate ); // validate and store commission rate

        System.out.printf(
            "%nCommissionEmployee4 constructor:\n%s\n", this);
    } // end five-argument CommissionEmployee4 constructor
27    // set first name
28    public void setFirstName( String first )
29    {
30       firstName = first;
31    } // end method setFirstName
32
33    // return first name
34    public String getFirstName()
35    {
36       return firstName;
37    } // end method getFirstName
38
39    // set last name
40    public void setLastName( String last )
41    {
42       lastName = last;
43    } // end method setLastName
44
45    // return last name
46    public String getLastName()
47    {
48       return lastName;
49    } // end method getLastName
50
51    // set social security number
52    public void setSocialSecurityNumber( String ssn )
53    {
54       socialSecurityNumber = ssn; // should validate
55    } // end method setSocialSecurityNumber
56
// 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
```java
public double getCommissionRate()
{
    return commissionRate;
} // end method getCommissionRate

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

public String toString()
{
    return String.format("%s: %s
%s: %s
%s: %.2f
%s: %.2f",
    "commission employee", getFirstName(), getLastName(),
    "social security number", getSocialSecurityNumber(),
    "gross sales", getGrossSales(),
    "commission rate", getCommissionRate() );
} // end method toString
} // end class CommissionEmployee4
```
public class BasePlusCommissionEmployee5 extends CommissionEmployee4
{
    private double baseSalary; // base salary per week

    // six-argument constructor
    public BasePlusCommissionEmployee5( 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

        System.out.printf( "\nBasePlusCommissionEmployee5 constructor: \n%sn", this );
    } // end six-argument BasePlusCommissionEmployee5 constructor

    // set base salary
    public void setBaseSalary( double salary )
    {
        baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
    } // end method setBaseSalary
25    // return base salary
26    public double getBaseSalary()
27    {
28       return baseSalary;
29    } // end method getBaseSalary
30
31    // calculate earnings
32    public double earnings()
33    {
34       return getBaseSalary() + super.earnings();
35    } // end method earnings
36
37    // return String representation of BasePlusCommissionEmployee5
38    public String toString()
39    {
40       return String.format( "%s %s
%s: %.2f", "base-salaried",
41                                  super.toString(), "base salary", getBaseSalary() );
42    } // end method toString
43 } // end class BasePlusCommissionEmployee5
// 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
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.
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
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clone</td>
<td>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 <code>clone</code> as a public method and should implement interface <code>Cloneable</code> (package <code>java.lang</code>). 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 <code>clone</code> 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 <code>clone</code>. You can learn more about cloning in the following article: java.sun.com/developer/JDCTechTips/2001/tt0306.html</td>
</tr>
</tbody>
</table>

**Fig. 9.18** | **Object methods that are inherited directly or indirectly by all classes. (Part 1 of 4)**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| `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)**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>finalize</td>
<td>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.</td>
</tr>
<tr>
<td>getClass</td>
<td>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.</td>
</tr>
</tbody>
</table>
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
// 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" );
    }
}
Lines 34-36

Attach the labels to the JFrame at north, center and south
Fig. 9.20 | JLabel displaying shape statistics.