Relationships among Classes

Java™ How to Program, 10/e
Late Objects Version

References & Reading

- The content is mainly selected (sometimes modified) from the original slides provided by the authors of the textbook

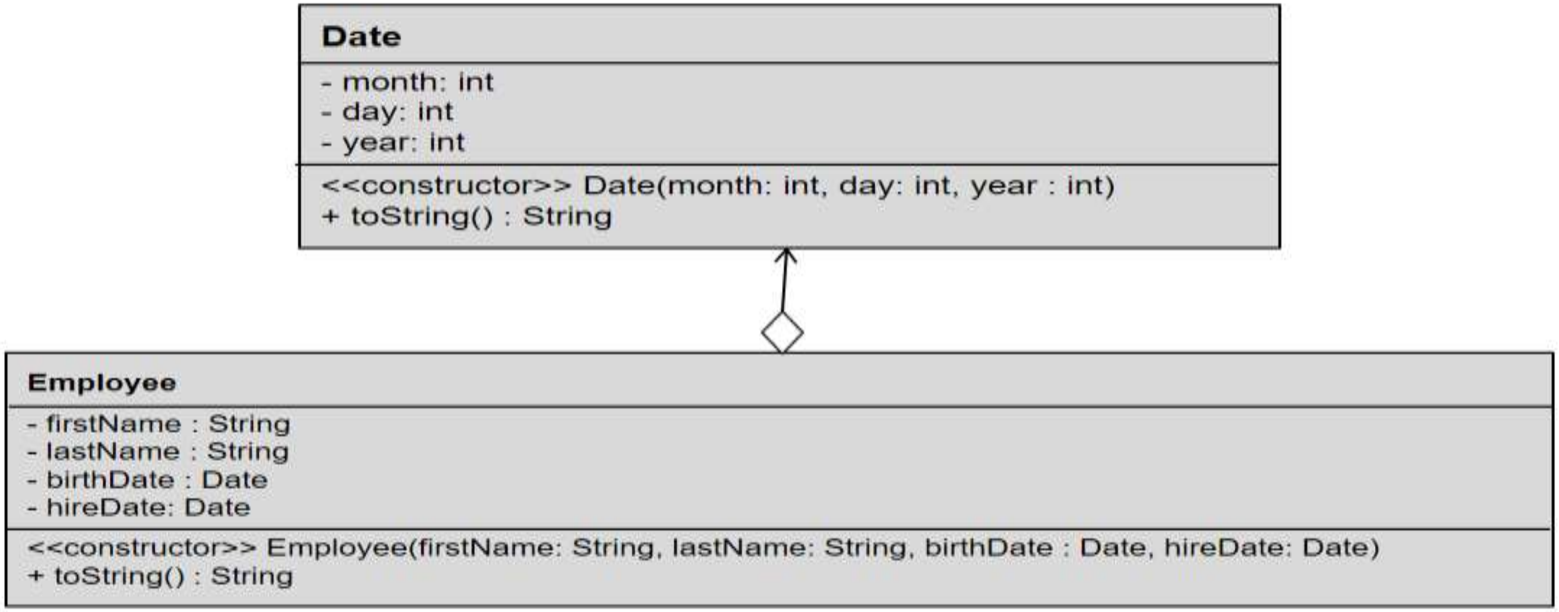
- Readings
  - Chapter 8: Classes and Objects: A Deeper Look
  - Chapter 9: Object-Oriented Programming: Inheritance

Outline

- Composition
- Introduction
- Superclasses and Subclasses
- protected Members
- Relationship Between Superclasses and Subclasses
- Constructors in Subclasses
- Class Object

Composition

- A class can have references to objects of other classes as members.
- This is called composition and is sometimes referred to as a *has-a* relationship.
- The following example presents a class `Date` (Fig. 8.7), a class `Employee` (Fig. 8.8) that *has* 2 Dates, and a class `EmployeeTest` (Fig. 8.9) to demonstrate the composition.
- UML representation of Composition: ←————◇



Date

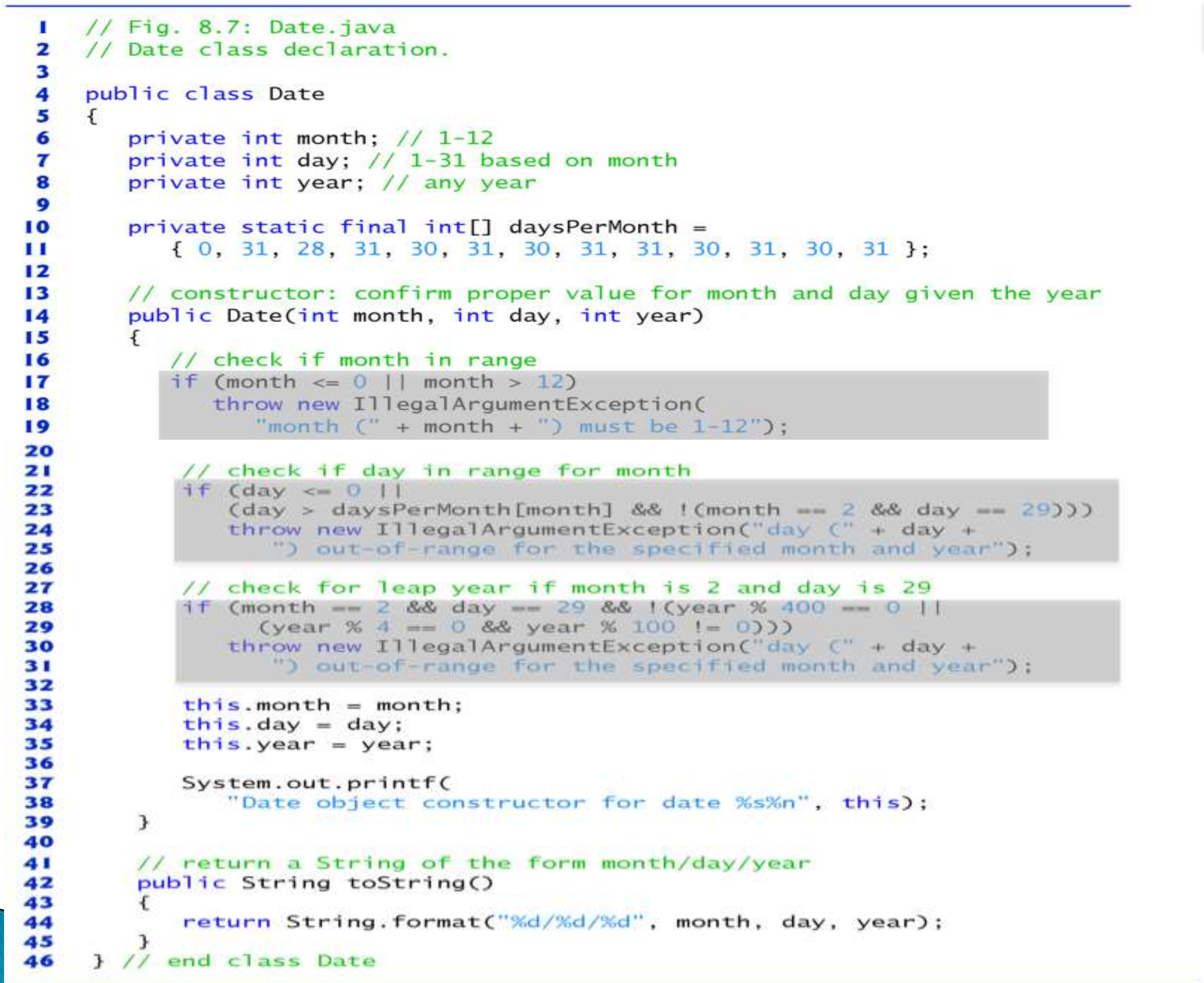
- month: int
- day: int
- year: int

<<constructor>> Date(month: int, day: int, year : int)
+ toString() : String

Employee

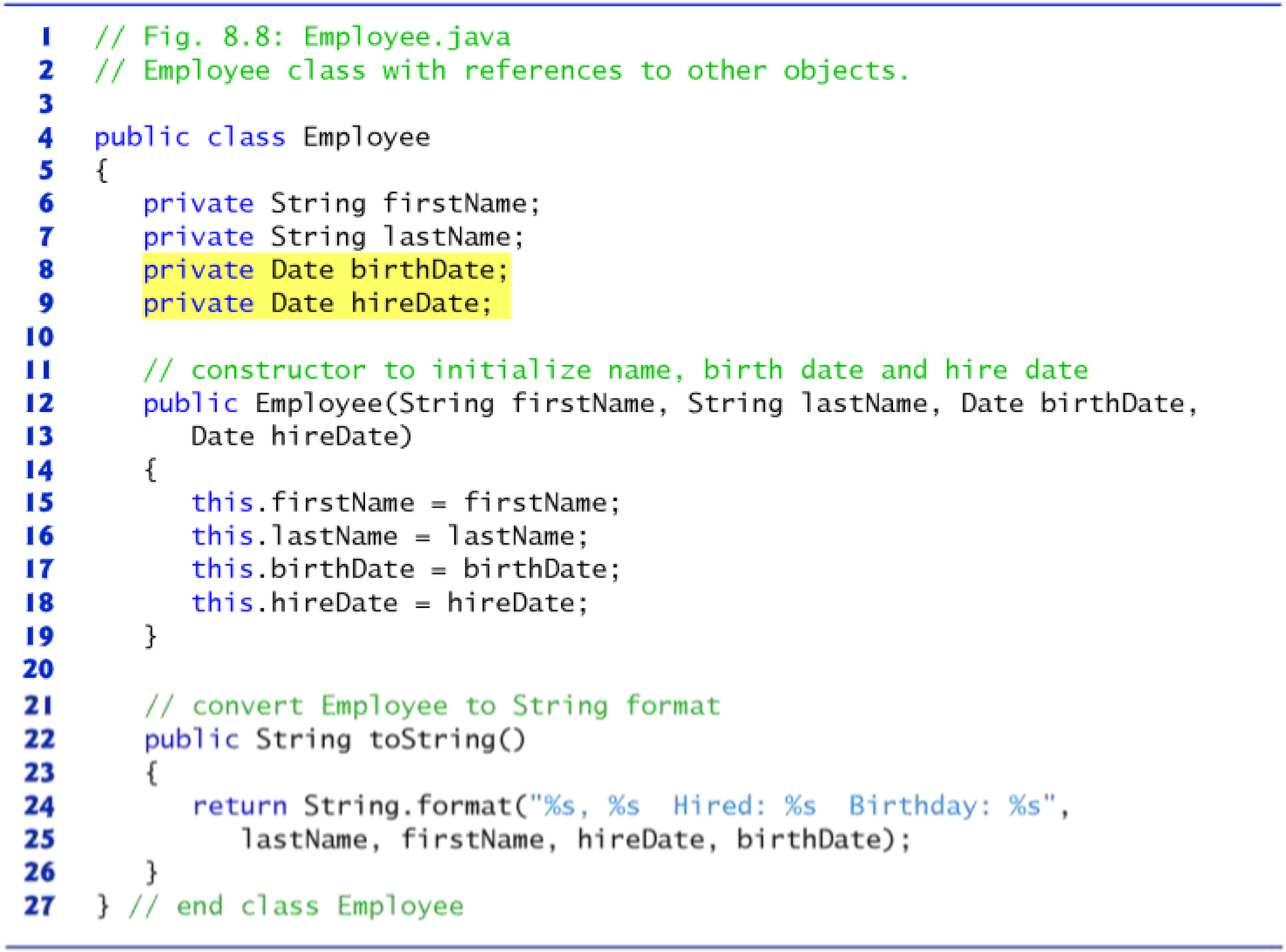
- firstName : String
- lastName : String
- birthDate : Date
- hireDate: Date

<<constructor>> Employee(firstName: String, lastName: String, birthDate : Date, hireDate: Date)
+ toString() : String



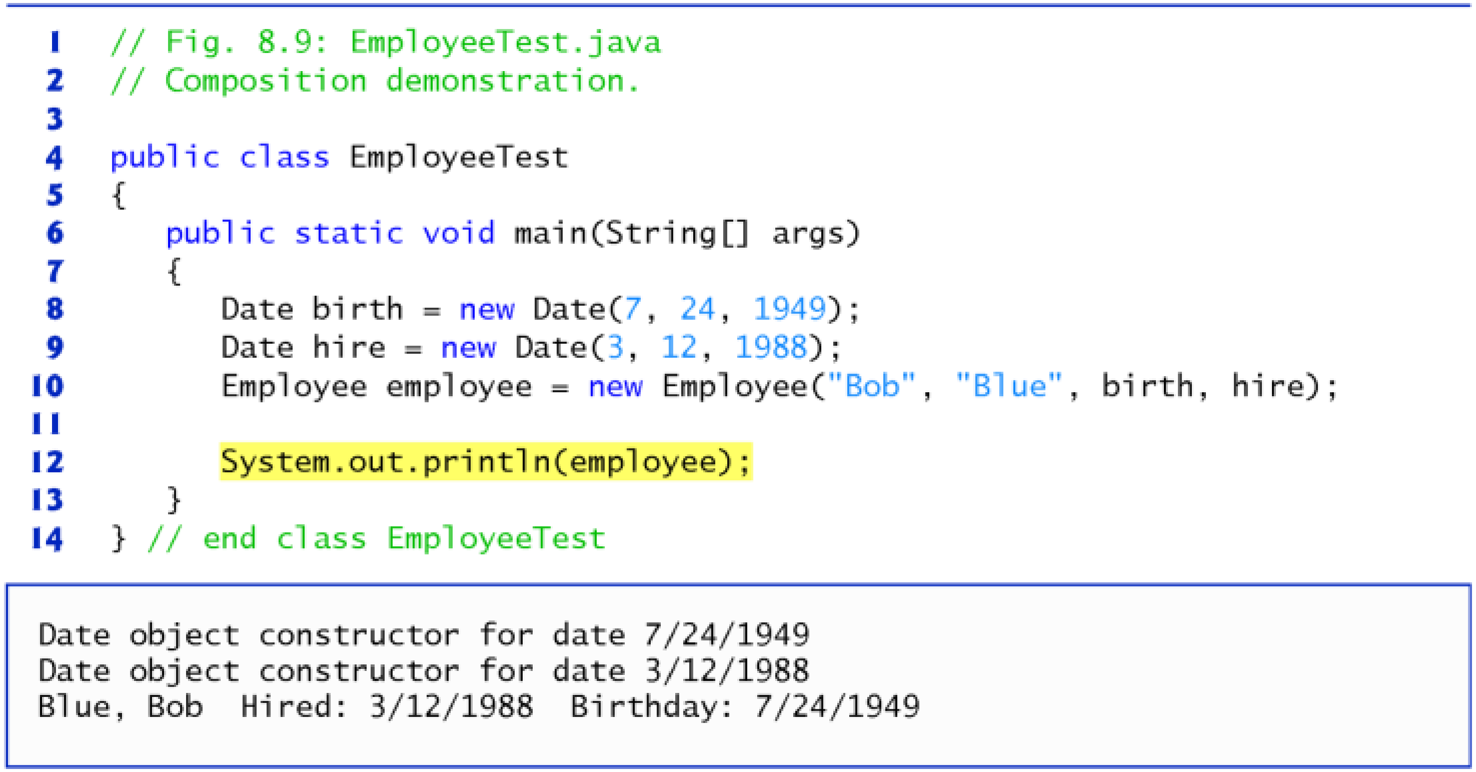
```java
// Fig. 8.7: Date.java
// Date class declaration.

public class Date
{
   private int month; // 1-12
   private int day; // 1-31 based on month
   private int year; // any year

   private static final int[] daysPerMonth =
      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   // constructor: confirm proper value for month and day given the year
   public Date(int month, int day, int year)
   {
      // check if month in range
      if (month <= 0 || month > 12)
         throw new IllegalArgumentException(
            "month (" + month + ") must be 1-12");

      // check if day in range for month
      if (day <= 0 ||
         (day > daysPerMonth[month] && !(month == 2 && day == 29)))
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");

      // check for leap year if month is 2 and day is 29
      if (month == 2 && day == 29 && !(year % 400 == 0 ||
           (year % 4 == 0 && year % 100 != 0)))
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");

      this.month = month;
      this.day = day;
      this.year = year;

      System.out.printf(
         "Date object constructor for date %s%n", this);
   }

   // return a String of the form month/day/year
   public String toString()
   {
      return String.format("%d/%d/%d", month, day, year);
   }
} // end class Date
```

Fig. 8.7 | Date class declaration. (Part 3 of 3.)



```java
// Fig. 8.8: Employee.java
// Employee class with references to other objects.

public class Employee
{
   private String firstName;
   private String lastName;
   private Date birthDate;
   private Date hireDate;

   // constructor to initialize name, birth date and hire date
   public Employee(String firstName, String lastName, Date birthDate,
      Date hireDate)
   {
      this.firstName = firstName;
      this.lastName = lastName;
      this.birthDate = birthDate;
      this.hireDate = hireDate;
   }

   // convert Employee to String format
   public String toString()
   {
      return String.format("%s, %s  Hired: %s  Birthday: %s",
         lastName, firstName, hireDate, birthDate);
   }
} // end class Employee
```

Fig. 8.8 | Employee class with references to other objects.



```java
// Fig. 8.9: EmployeeTest.java
// Composition demonstration.

public class EmployeeTest
{
   public static void main(String[] args)
   {
      Date birth = new Date(7, 24, 1949);
      Date hire = new Date(3, 12, 1988);
      Employee employee = new Employee("Bob", "Blue", birth, hire);

      System.out.println(employee);
   }
} // end class EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Fig. 8.9 | Composition demonstration.

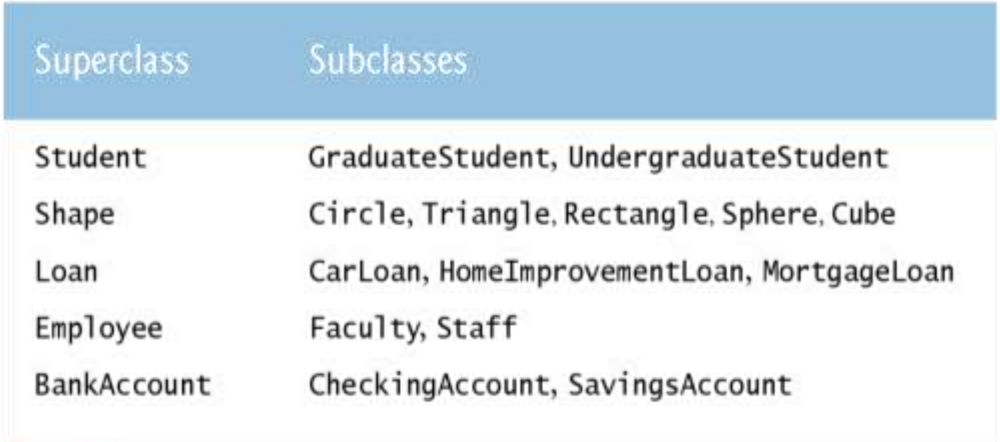
Introduction to inheritance

- Inheritance
  - A new class (subclass) is created by acquiring an existing class's (superclass) members and possibly embellishing them with new or modified capabilities.
  - Can save time during program development by basing new classes on existing proven and debugged high-quality software.
  - Increases the likelihood that a system will be implemented and maintained effectively.
  - Java supports only single inheritance, in which each class is derived from exactly one direct superclass.

- Inheritance is sometimes referred to as a *is-a* relationship.
- A subclass can be a superclass of future subclasses.
- A subclass can add its own fields and methods (becomes more specific than its superclass)

Superclasses and Subclasses

- A direct superclass is the superclass from which the subclass explicitly inherits.
- An indirect superclass is any class above the direct superclass in the class hierarchy.
- The Java class hierarchy begins with class `Object` (in package `java.lang`)
  - *Every* class in Java directly or indirectly extends (or "inherits from") `Objec`
- Figure 9.1 lists several simple examples of superclasses and subclasses
  - Superclasses tend to be "more general" and subclasses "more specific."



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

Fig. 9.1 | Inheritance examples.

Superclasses and Subclasses (Cont.)

- A superclass exists in a hierarchical relationship with its subclasses.
- Fig. 9.2 shows a sample university community class hierarchy
  - Also called an inheritance hierarchy.
- Each arrow in the hierarchy represents an *is-a relationship*.
- Follow the arrows upward in the class hierarchy
  - “an Employee *is a* CommunityMember”
  - “a Teacher is a Faculty member”
- CommunityMember is the direct superclass of Employee, Student and Alumnus, and is an indirect superclass of all the other classes in the diagram.

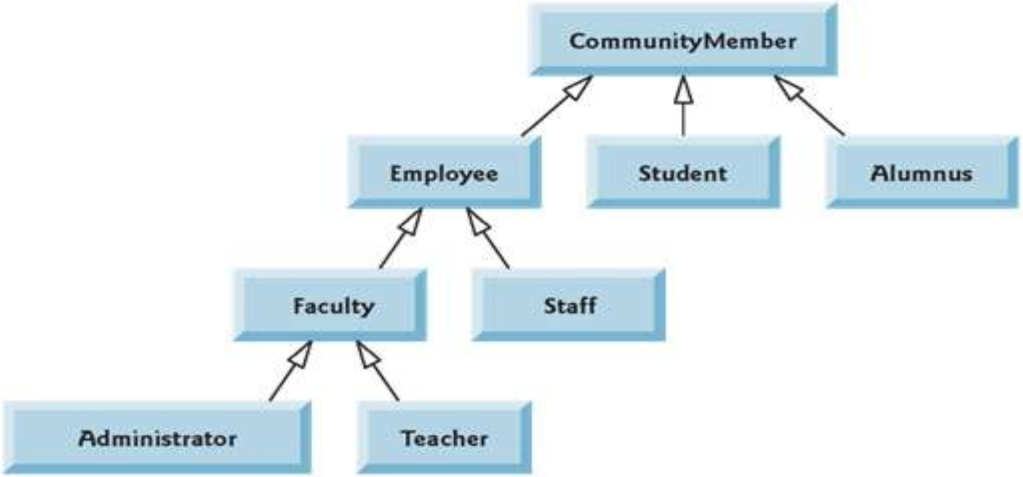


Fig. 9.2 | Inheritance hierarchy UML class diagram for university CommunityMembers.

Superclasses and Subclasses (Cont.)

- Objects of all classes that extend a common superclass can be treated as objects of that superclass.
  - Commonality expressed in the members of the superclass.
- Inheritance issue
  - All members of all superclasses are inherited (except private)
  - A subclass can inherit methods that it does not need or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.
  - The subclass can override (redefine) the superclass method with an appropriate implementation.

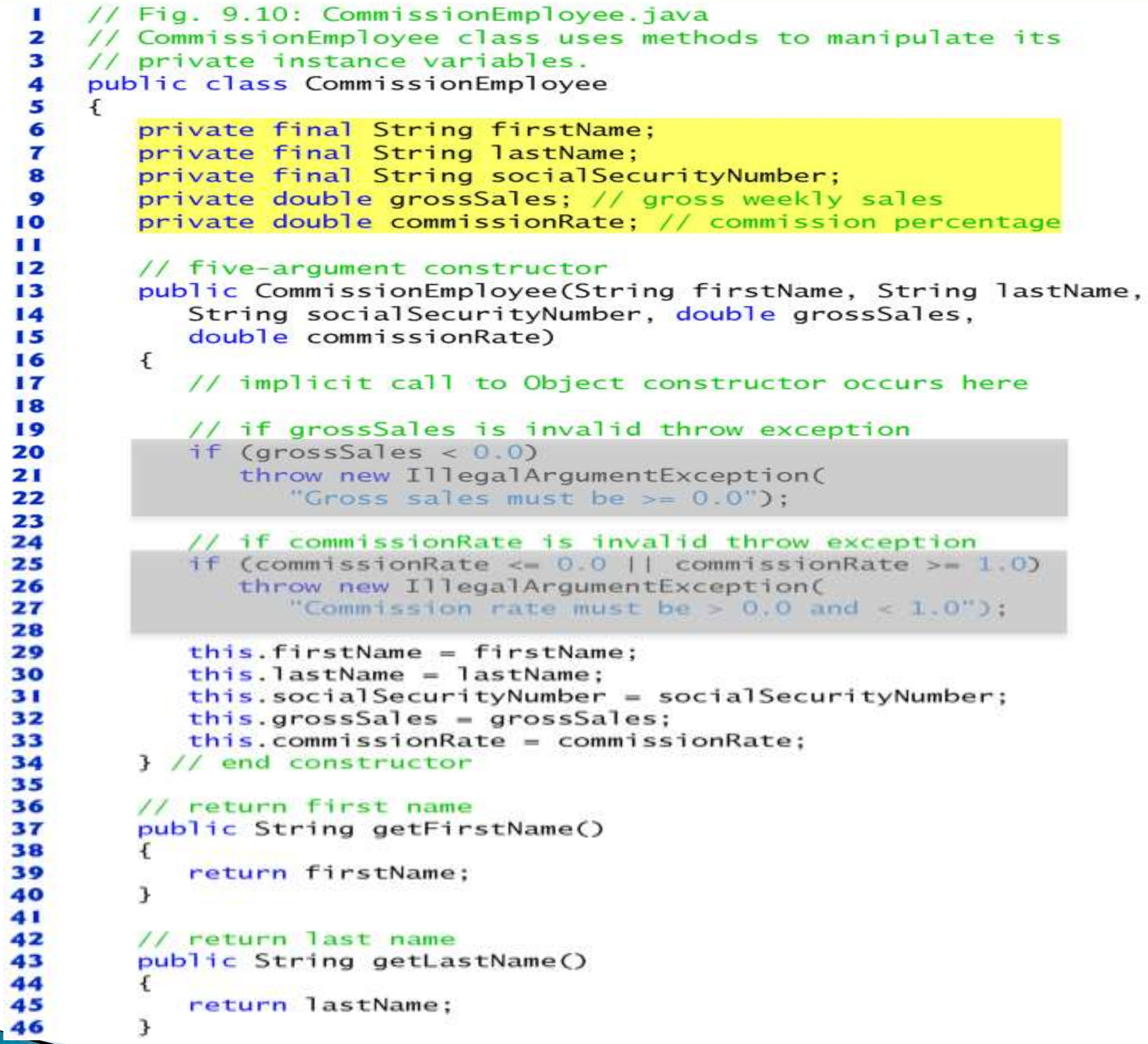
protected Members

- A class's `public` members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A superclass's `private` members are accessible only within the class itself but they are *hidden* from its subclasses (can be accessed only through the `public` or `protected` methods inherited from the superclass)
- `protected` access is an intermediate level of access between `public` and `private`.
  - A superclass's `protected` members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the *same package*
  - All `public` and `protected` superclass members retain their original access modifier when they become members of the subclass.
- Subclass methods can refer to `public` and `protected` members inherited from the superclass simply by using the member names.
- When a subclass method *overrides* an inherited superclass method, the *superclass* version of the method can be accessed from the *subclass* by preceding the superclass method name with keyword `super` and a dot (`.`) separator.

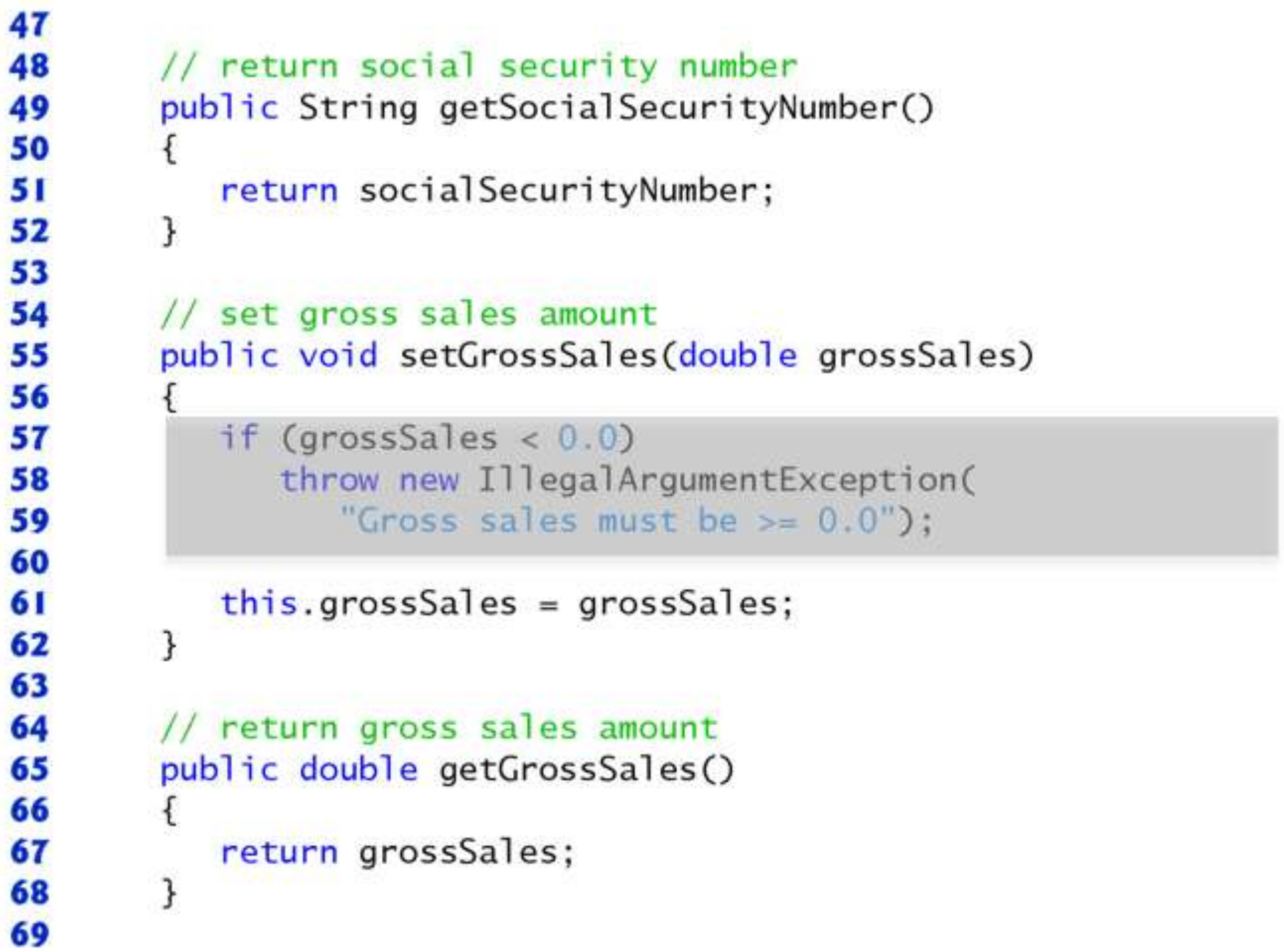
Creating and Using a CommissionEmployee Class

- Class `CommissionEmployee` (Fig. 9.10) extends implicitly class Object (from package `java.lang`).
  - `CommissionEmployee` inherits `Object`'s methods.
  - If you don't explicitly specify which class a new class extends, the class extends `Object` implicitly.

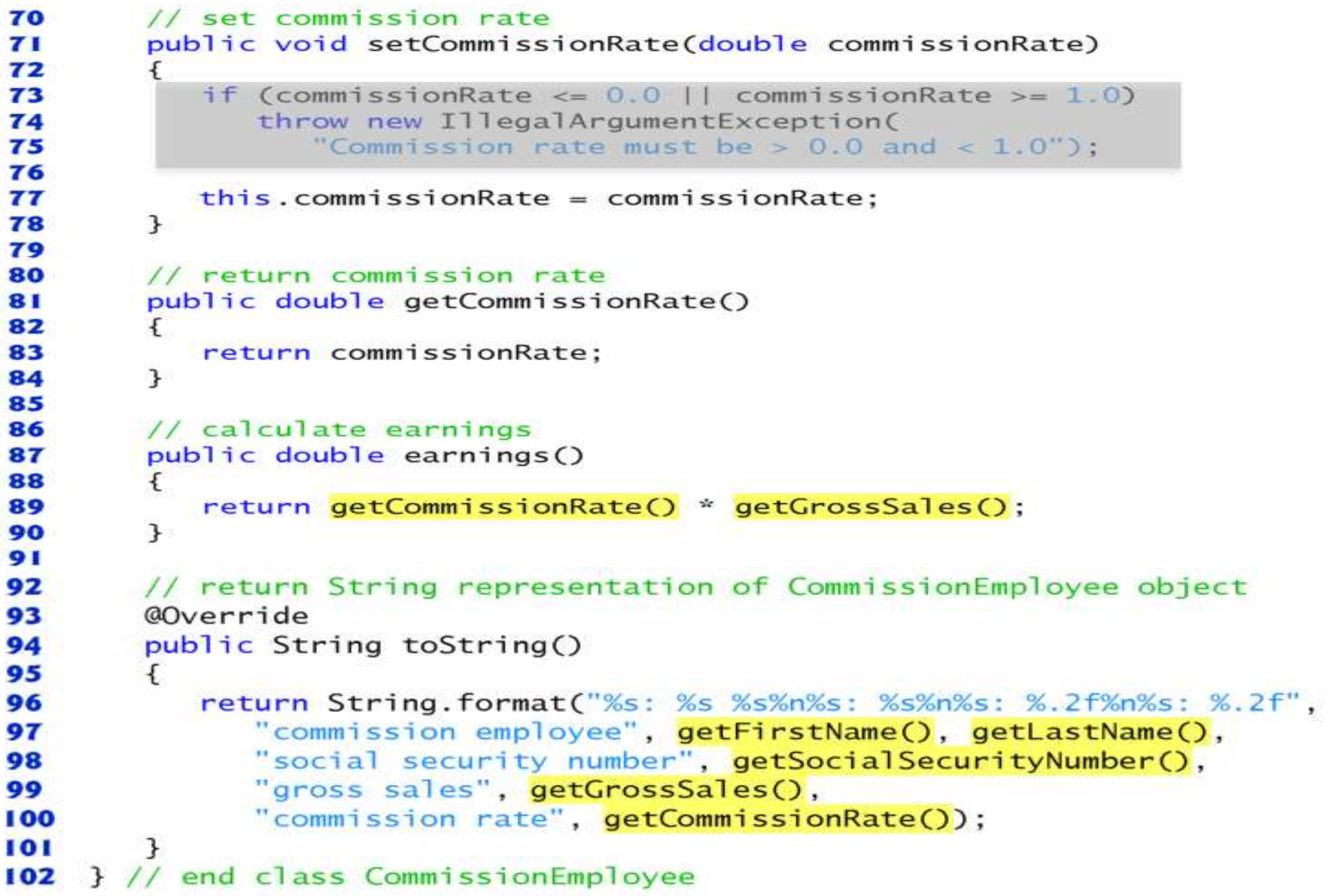
- Constructors are *not* inherited.
- *The first task of a subclass constructor is to call its direct superclass's constructor* explicitly (super) or implicitly
- If the code does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default or no-argument constructor.



```java
// Fig. 9.10: CommissionEmployee.java
// CommissionEmployee class uses methods to manipulate its
// private instance variables.
public class CommissionEmployee
{
   private final String firstName;
   private final String lastName;
   private final String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee(String firstName, String lastName,
      String socialSecurityNumber, double grossSales,
      double commissionRate)
   {
      // implicit call to Object constructor occurs here

      // if grossSales is invalid throw exception
      if (grossSales < 0.0)
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0");

      // if commissionRate is invalid throw exception
      if (commissionRate <= 0.0 || commissionRate >= 1.0)
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0");

      this.firstName = firstName;
      this.lastName = lastName;
      this.socialSecurityNumber = socialSecurityNumber;
      this.grossSales = grossSales;
      this.commissionRate = commissionRate;
   } // end constructor

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

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



```java

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

   // set gross sales amount
   public void setGrossSales(double grossSales)
   {
      if (grossSales < 0.0)
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0");

      this.grossSales = grossSales;
   }

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

```



```java
      // set commission rate
      public void setCommissionRate(double commissionRate)
      {
         if (commissionRate <= 0.0 || commissionRate >= 1.0)
            throw new IllegalArgumentException(
               "Commission rate must be > 0.0 and < 1.0");

         this.commissionRate = commissionRate;
      }

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

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

      // return String representation of CommissionEmployee object
      @Override
      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 class CommissionEmployee
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its private instance variables. (Part 5 of 5.)

Creating & Using a CommissionEmployee Class (Cont.)

- `toString` is one of the methods that *every* class inherits directly or indirectly from class `Object`.
  - Returns a `String` representing an object.
  - Called implicitly whenever an object must be converted to a `String` representation.

- Class `Object`'s `toString` method returns a `String` that includes the name of the object's class.
  - This is primarily a placeholder that can be overridden by a subclass to specify an appropriate `String` representation.

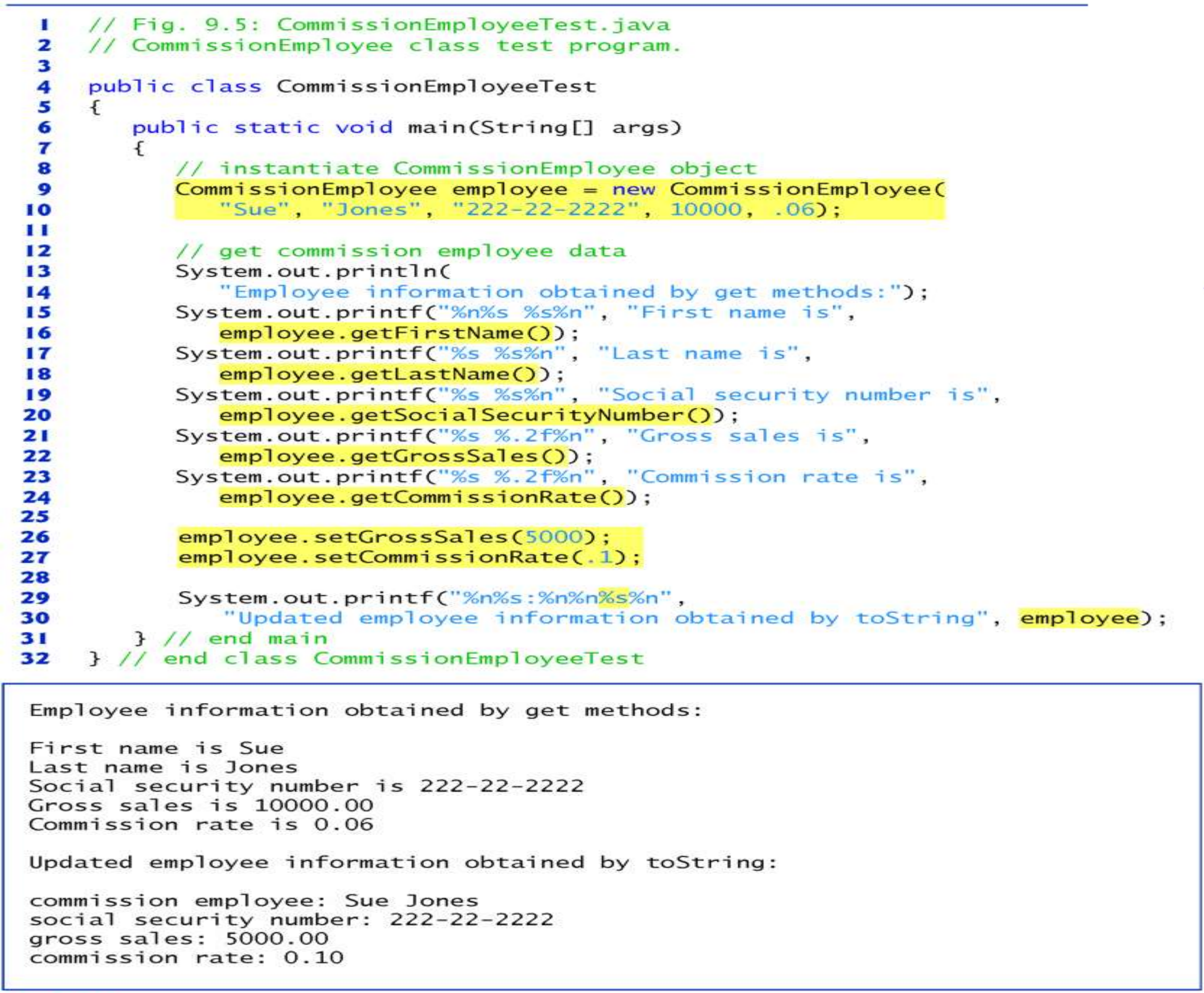
- To override a superclass method, a subclass must declare a method with the same signature as the superclass method

Creating & Using a CommissionEmployee Class (Cont.)

- The optional `@Override` annotation
  - Indicates/ensures that a method should override a superclass method with the same signature.
  - If it does not, a compilation error occurs.
- The `CommissionEmployee` class overrides `Object`’s `toString` method.

Changing inherited method access modifier

- A subclass can change the access modifier of an inherited method
- It is a compilation error to override a method with a more restricted access modifier
  - A public superclass method cannot become a protected or private subclass method
  - A protected superclass method cannot become a private subclass method



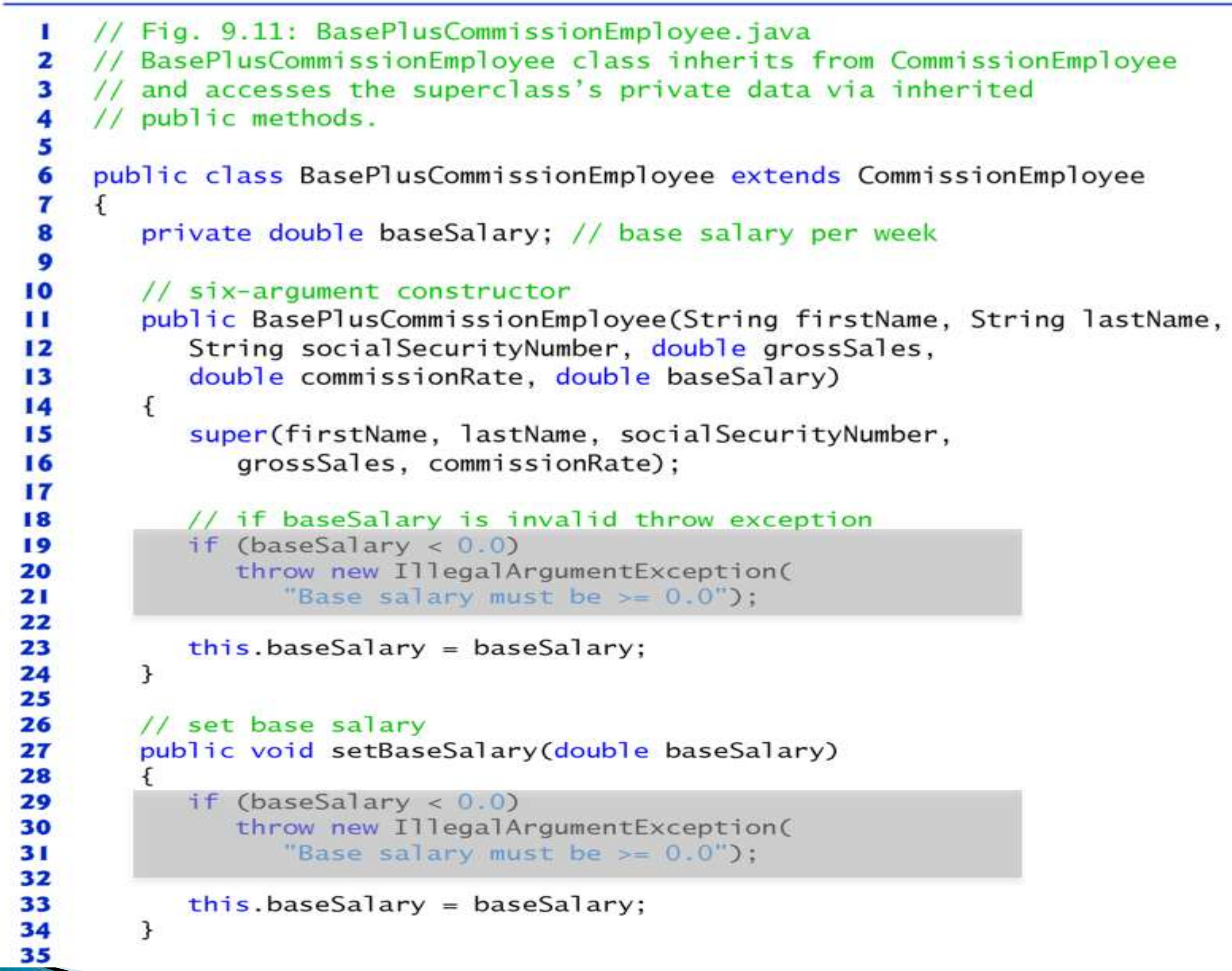
```java
// Fig. 9.5: CommissionEmployeeTest.java
// CommissionEmployee class test program.

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:");
      System.out.printf("%n%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(5000);
      employee.setCommissionRate(.1);

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

```
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: 5000.00
commission rate: 0.10
```

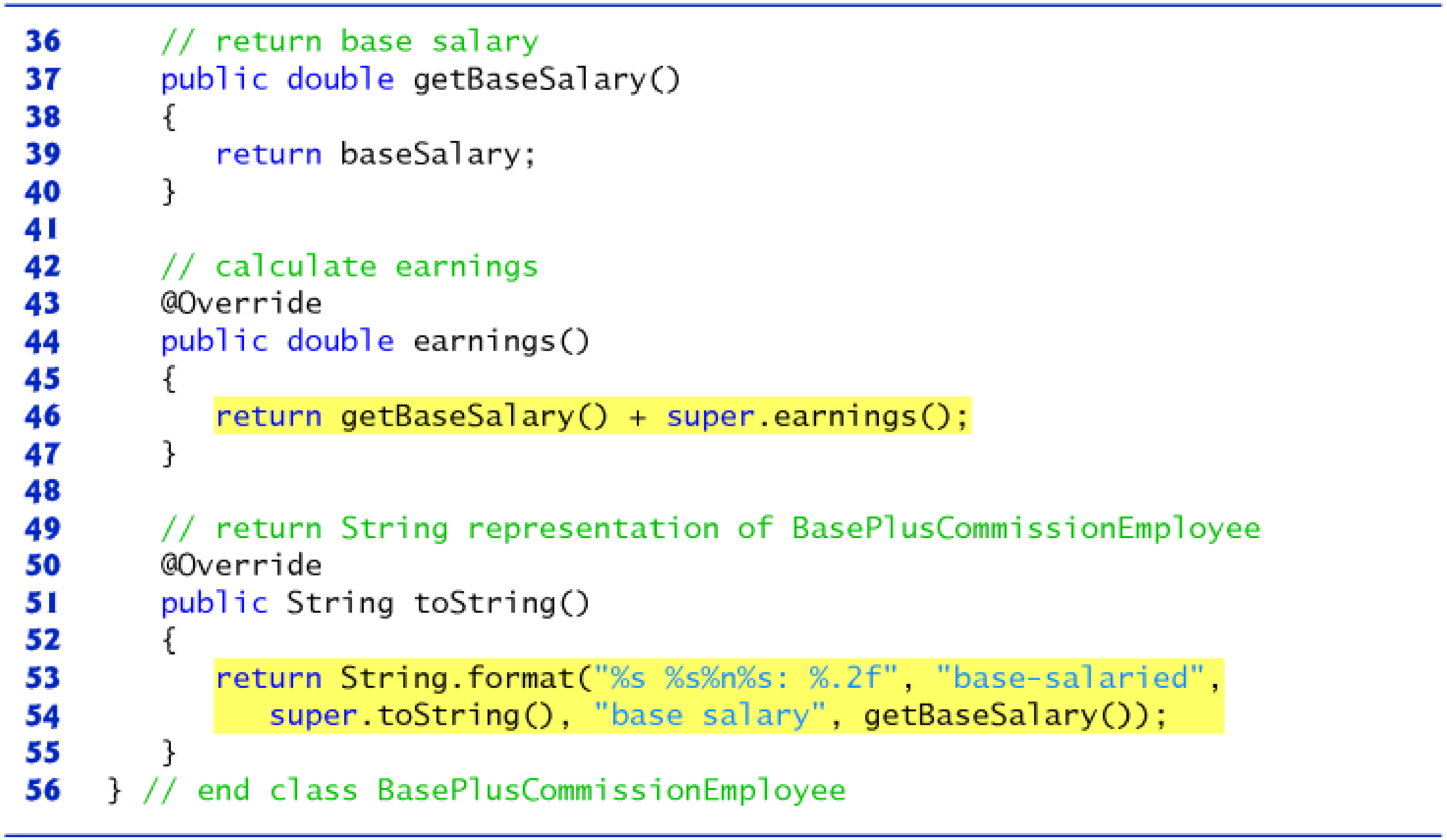
Fig. 9.5 | CommissionEmployee class test program. (Part 2 of 2.)

Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy

- Class `BasePlusCommissionEmployee` class *extends* class `CommissionEmployee.`
- A `BasePlusCommissionEmployee` object *is a* `CommissionEmployee`
  - Inheritance passes on class `CommissionEmployee`’s capabilities.
- Class `BasePlusCommissionEmployee` also has instance variable `baseSalary`.
- Subclass `BasePlusCommissionEmployee` inherits `CommissionEmployee`’s instance variables and methods
  - Only `CommissionEmployee`’s `public` and `protected` members are directly accessible in the subclass.



```java
// Fig. 9.11: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class inherits from CommissionEmployee
// and accesses the superclass's private data via inherited
// public methods.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee(String firstName, String lastName,
      String socialSecurityNumber, double grossSales,
      double commissionRate, double baseSalary)
   {
      super(firstName, lastName, socialSecurityNumber,
         grossSales, commissionRate);

      // if baseSalary is invalid throw exception
      if (baseSalary < 0.0)
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0");

      this.baseSalary = baseSalary;
   }

   // set base salary
   public void setBaseSalary(double baseSalary)
   {
      if (baseSalary < 0.0)
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0");

      this.baseSalary = baseSalary;
   }

```

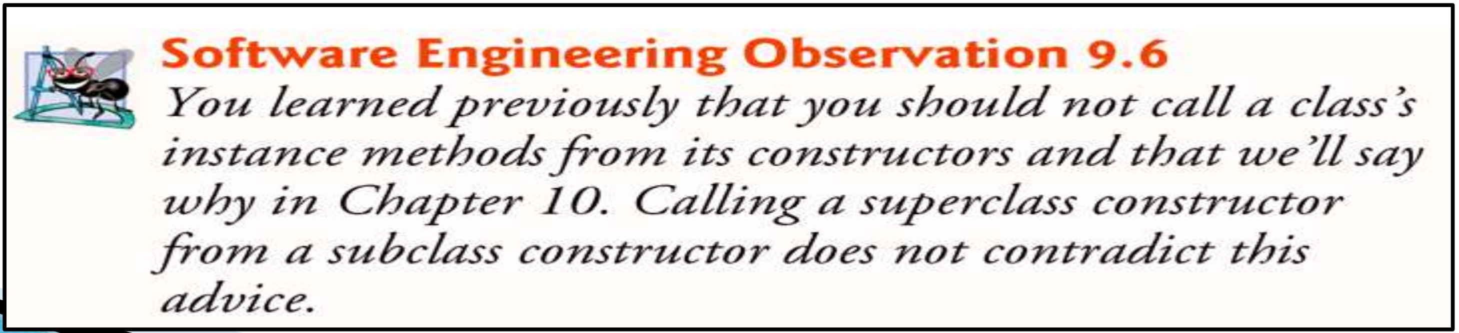


```java
   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   }

   // calculate earnings
   @Override
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   }

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

Fig. 9.11 | BasePlusCommissionEmployee class inherits from CommissionEmployee and accesses the superclass's private data via inherited public methods. (Part 3 of 3.)

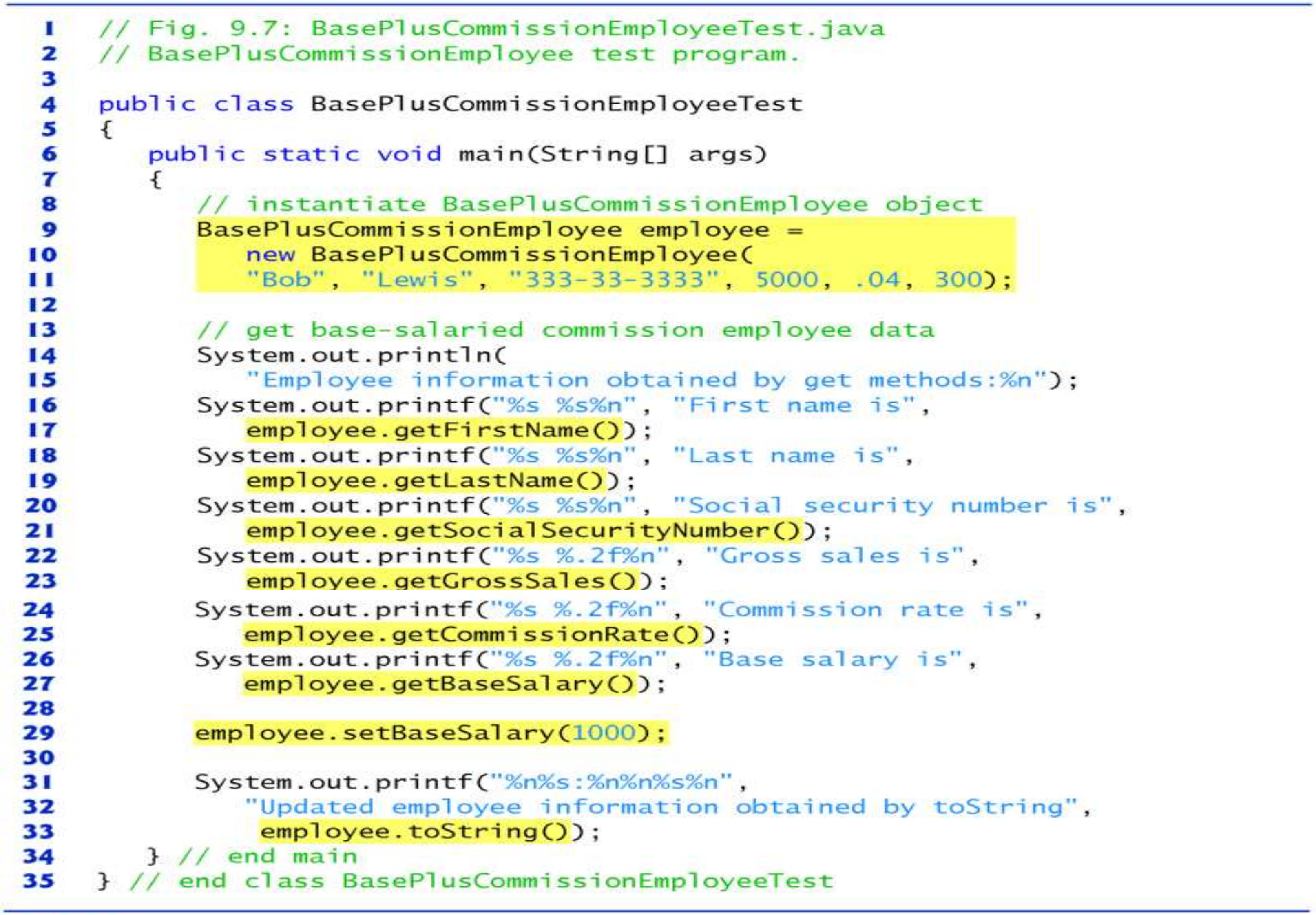
Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)

- Each subclass constructor must implicitly or explicitly call one of its superclass's constructors to initialize the instance variables inherited from the superclass.
  - Superclass constructor call syntax—keyword `super`, followed by a set of parentheses containing the superclass constructor arguments.
  - Must be the *first* statement in the constructor's body.
- If the subclass constructor did not invoke the superclass's constructor explicitly, the compiler would attempt to insert a call to the superclass's default or no-argument constructor.
  - Class `CommissionEmployee` does not have such a constructor, so the compiler would issue an error.
- You can explicitly use `super()` to call the superclass's no-argument or default constructor, but this is rarely done.



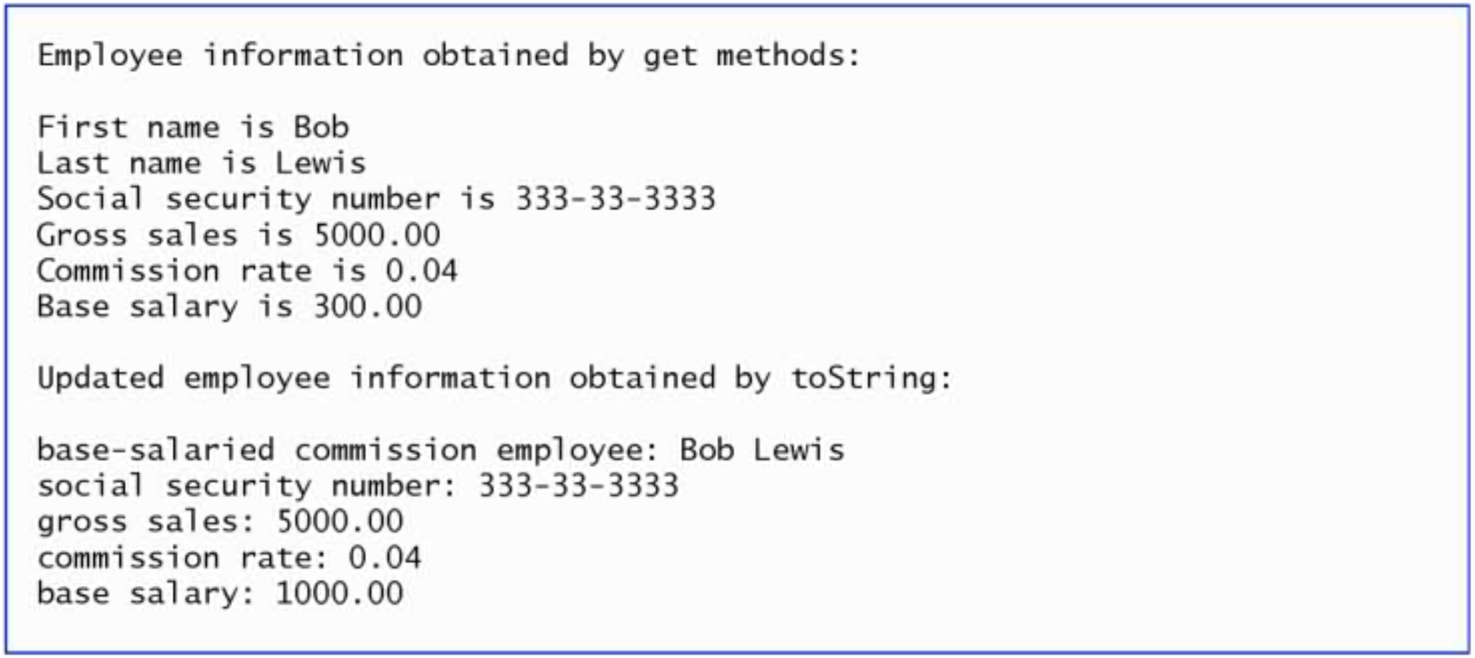
Software Engineering Observation 9.6

You learned previously that you should not call a class's instance methods from its constructors and that we'll say why in Chapter 10. Calling a superclass constructor from a subclass constructor does not contradict this advice.



```java
// Fig. 9.7: BasePlusCommissionEmployeeTest.java
// BasePlusCommissionEmployee test program.

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);

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

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 2 of 3.)



```
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
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 3 of 3.)

CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables

- Methods `earnings` and `toString` each invoke their superclass versions and do not access instance variable.
- Method `earnings` overrides class the superclass's `earnings` method.

- Calls superclass's `earnings` method with `super.earnings()`.
  - Obtains the earnings based on commission alone.

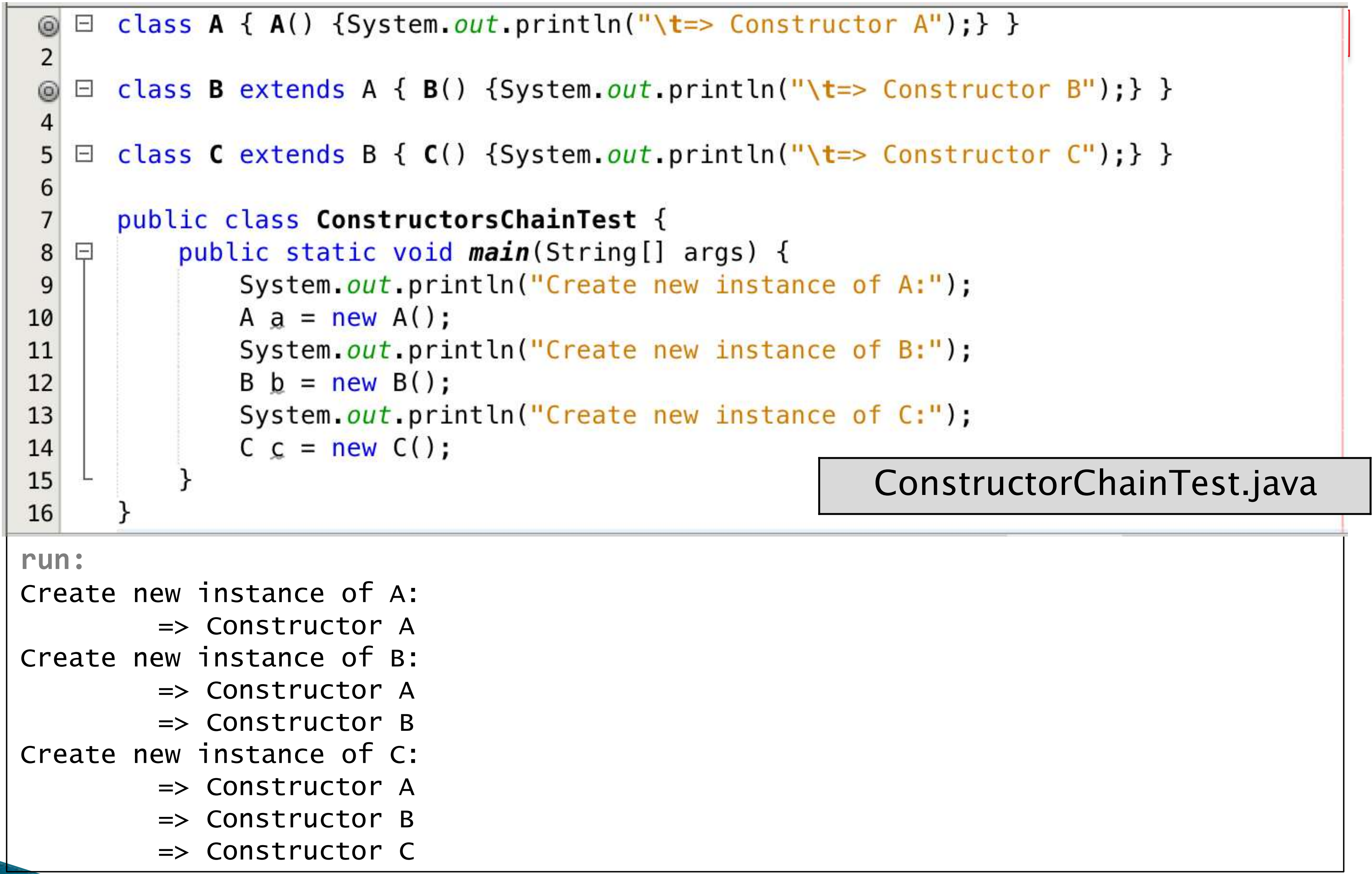
- Placing the keyword `super` and a dot (`.`) separator before the superclass method name invokes the superclass version of an overridden method.

- `BasePlusCommissionEmployee`'s `toString` method overrides class `CommissionEmployee`'s `toString` method.

- It creates part of the `String` representation by calling CommissionEmployee's toString method with the expression `super.toString()`.

Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - The subclass constructor, before performing its own tasks, explicitly uses `super` to call one of the constructors in its direct superclass or implicitly calls the superclass's default or no-argument constructor
- If the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up the hierarchy, and so on.
- The last constructor called in the chain is *always* `Object`'s constructor.
- Original subclass constructor's body finishes executing *last*.



```java
class A { A() {System.out.println("\t=> Constructor A");} }

class B extends A { B() {System.out.println("\t=> Constructor B");} }

class C extends B { C() {System.out.println("\t=> Constructor C");} }

public class ConstructorsChainTest {
    public static void main(String[] args) {
        System.out.println("Create new instance of A:");
        A a = new A();
        System.out.println("Create new instance of B:");
        B b = new B();
        System.out.println("Create new instance of C:");
        C c = new C();
    }
}
```

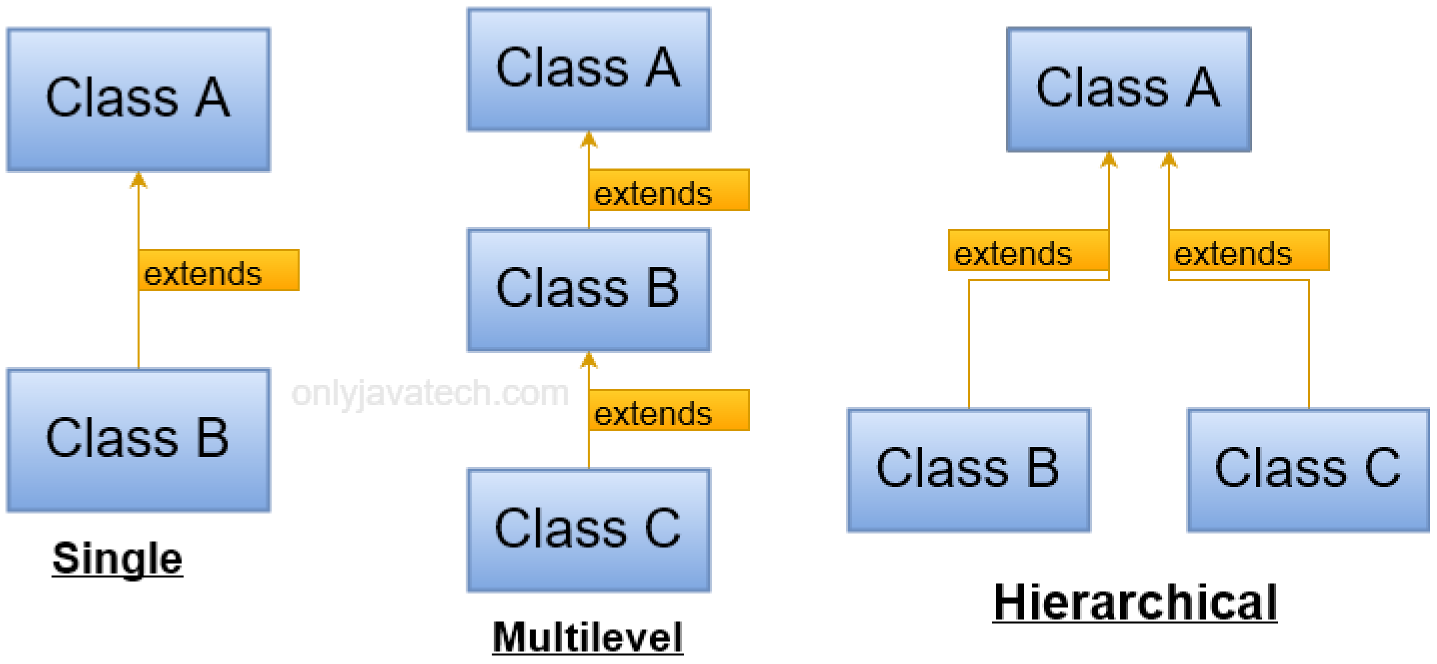
ConstructorChainTest.java

```
run:
Create new instance of A:
        => Constructor A
Create new instance of B:
        => Constructor A
        => Constructor B
Create new instance of C:
        => Constructor A
        => Constructor B
        => Constructor C
```

Extra slides

Superclasses and Subclasses

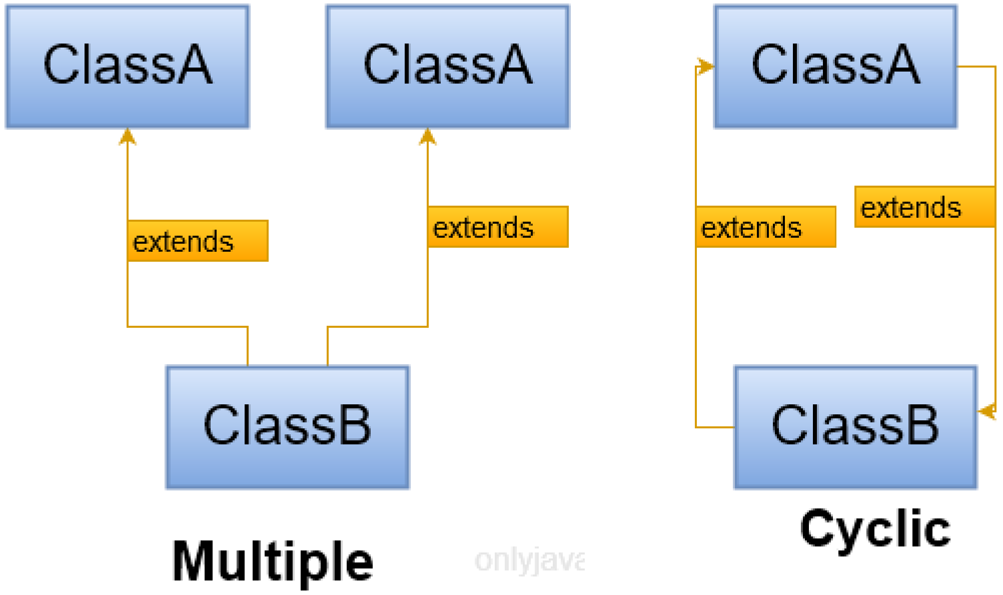
- Types of inheritance are supported in java



Reference:https://www.onlyjavatech.com/inheritance-in-java/

Superclasses and Subclasses (Cont.)

- Types of Inheritance are not supported in java.



Reference:https://www.onlyjavatech.com/inheritance-in-java/

Constructors in Subclasses

- Problem:
  - When a super-class has an argument constructor.
- Because:
  - Creating any object of subclass will (by default) invoke a default constructor of a super-class.
- Solutions:
  - At super-class: Add a default constructor.

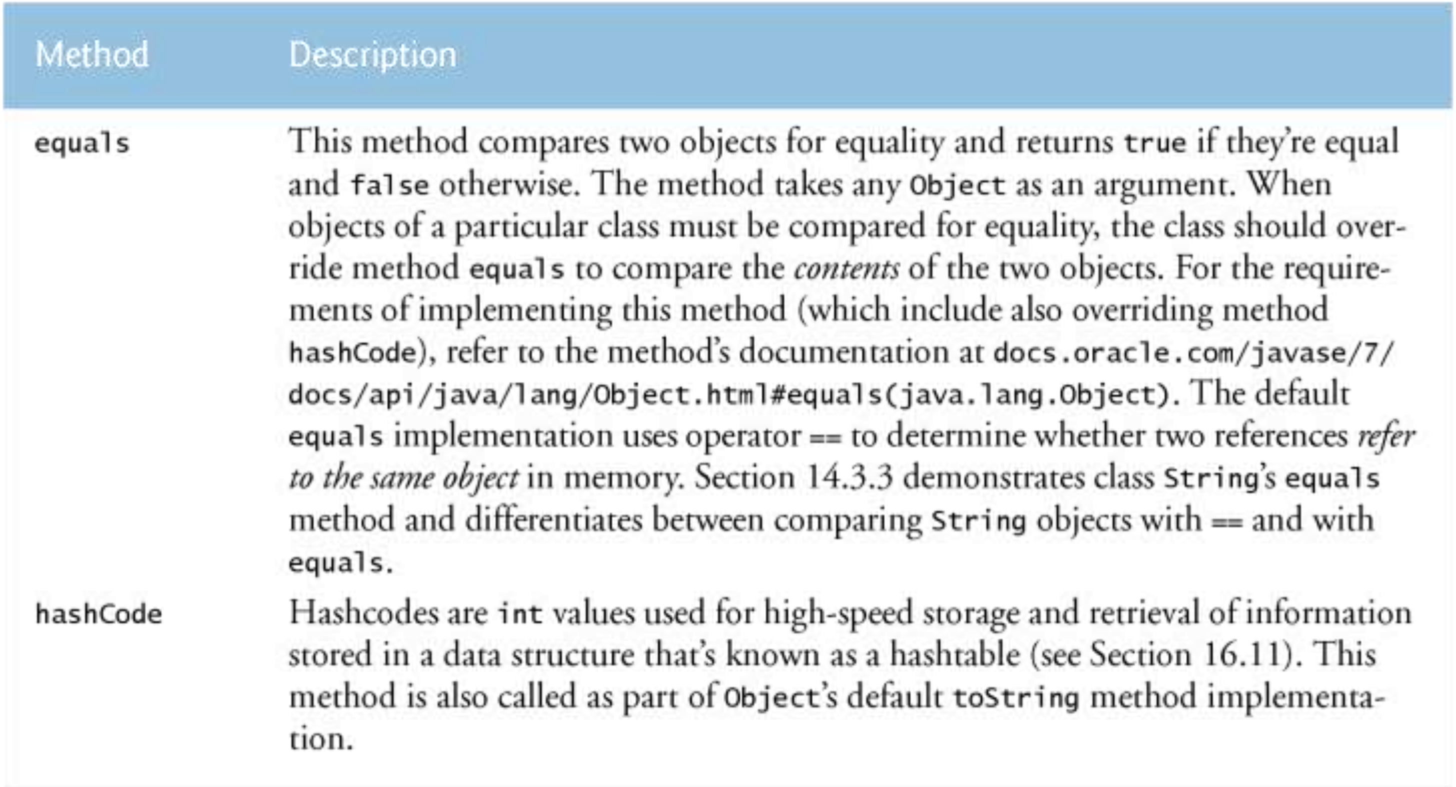
  OR

  - At subclass: Call explicitly an argument constructor of its direct super-class using *super()* in the first statement in a subclass constructor.

Self-Reading

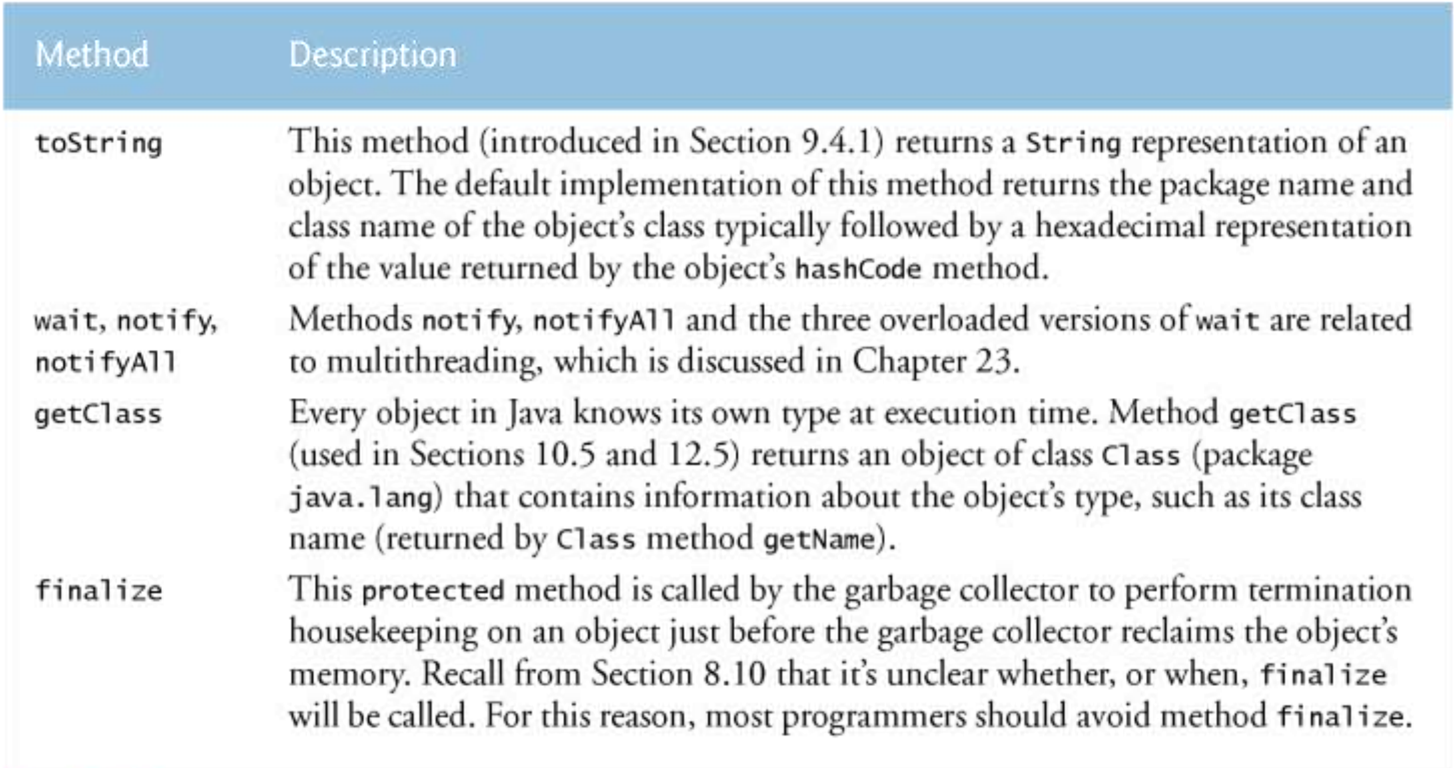
Class Object

- All classes in Java inherit directly or indirectly from class `Object`, so its 11 methods are inherited by all other classes.
- Figure 9.12 summarizes `Object`’s methods.
- Every array has an overridden `clone` method that copies the array.
  - If the array stores references to objects, the objects are not copied—a *shallow copy* is performed.



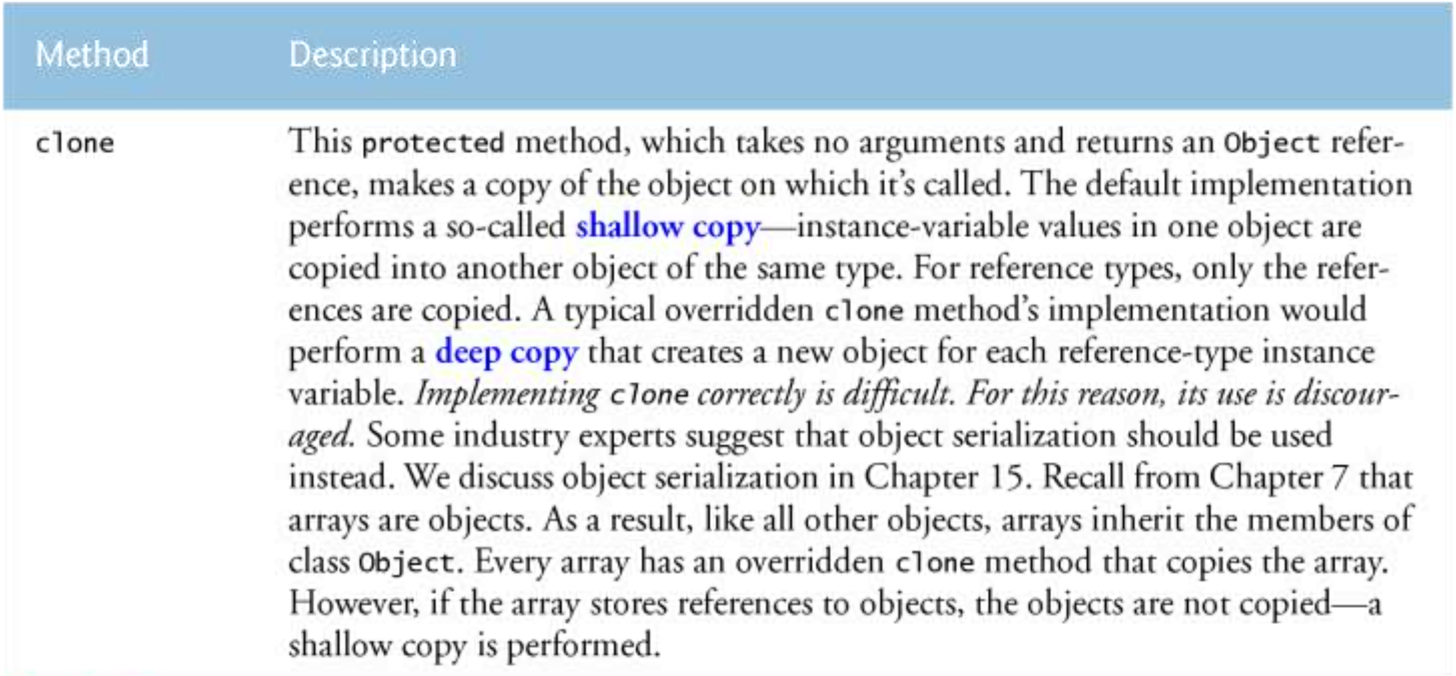
Method	Description
`equals`	This method compares two objects for equality and returns `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the *contents* of the two objects. For the requirements of implementing this method (which include also overriding method `hashCode`), refer to the method's documentation at `docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)`. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 14.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with == and with `equals`.
`hashCode`	Hashcodes are `int` values used for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (see Section 16.11). This method is also called as part of `Object`'s default `toString` method implementation.

Fig. 9.12 | `Object` methods. (Part 1 of 3.)



Method	Description
`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 typically followed by a hexadecimal representation of the value returned by the object's `hashCode` method.
`wait`, `notify`, `notifyAll`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Sections 10.5 and 12.5) 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`).
`finalize`	This `protected` method is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall from Section 8.10 that it's unclear whether, or when, `finalize` will be called. For this reason, most programmers should avoid method `finalize`.

Fig. 9.12 | `Object` methods. (Part 2 of 3.)



Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation 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. *Implementing `clone` correctly is difficult. For this reason, its use is discouraged.* Some industry experts suggest that object serialization should be used instead. We discuss object serialization in Chapter 15. Recall from Chapter 7 that arrays are objects. As a result, like all other objects, arrays inherit the members of class `Object`. Every array has an overridden `clone` method that copies the array. However, if the array stores references to objects, the objects are not copied—a shallow copy is performed.

Fig. 9.12 | `Object` methods. (Part 3 of 3.)