BBM 102 – Introduction to Programming II

Spring 2017

Inheritance

Instructors: Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi
TAs: Selma Dilek, Selim Yılmaz, Selman Bozkır

Today

- **Inheritance**
- **Notion of subclasses and superclasses**
- **protected members**
- **UML Class Diagrams for inheritance**

Inheritance

- A form of *software reuse* in which a new class is created by absorbing an existing class's members and 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.

Inheritance

- When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class.
  - Existing class is the *superclass*
  - New class is the *subclass*
- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
  - This is why inheritance is sometimes referred to as *specialization*.
- A subclass is more specific than its superclass and represents a more specialized group of objects.

Inheritance

- The *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") Object.
- Java supports only *single inheritance*, in which each class is derived from exactly one direct superclass.

Advantages of inheritance

- When a class inherits from another class, there are **three** benefits:

  (1) You can *reuse* the methods and data of the existing class

  (2) You can *extend* the existing class by adding new data and new methods

  (3) You can *modify* the existing class by overloading its methods with your own implementations

Relationships between classes

- We distinguish between the *is-a relationship* and the *has-a relationship*

- *Is-a* represents inheritance
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass

- *Has-a* represents composition
  - In a *has-a* relationship, an object contains as members references to other objects

Superclasses and Subclasses

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 tend to be "more general" and subclasses "more specific."

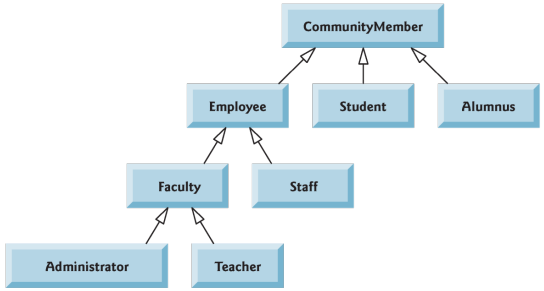


Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`.

- 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."

Superclasses and Subclasses (Cont.)

- Below is Shape inheritance hierarchy.
- Follow the arrows from the bottom of the diagram to the topmost superclass to identify several *is-a* relationships.
  - A Triangle *is a* TwoDimensionalShape and *is a* Shape
  - A Sphere *is a* ThreeDimensionalShape and *is a* Shape.

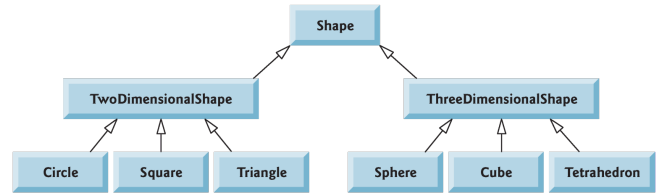


Fig. 9.3 | Inheritance hierarchy for Shapes.

Superclasses and Subclasses (Cont.)

- Not every class relationship is an inheritance relationship.
- *Has-a* relationship
  - Create classes by composition of existing classes.
  - Example: Given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`.
  - However, an `Employee` *has a* `BirthDate`, and an `Employee` *has a* `TelephoneNumber`.

`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 class's `private` members are accessible only within the class itself.
- `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
  - protected members also have package access.

`protected` Members (Cont.)

- A superclass's private members are hidden in its subclasses
  - They can be accessed only through the public or protected methods inherited from the superclass
- 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 method can be accessed from the subclass by preceding the superclass method name with keyword *super* and a dot (.) separator.

Case Study: Commission Employees

- Inheritance hierarchy containing types of employees in a company's payroll application
- Commission employees are paid a percentage of their sales
- Base-salaried commission employees receive a base salary plus a percentage of their sales.

Creating and Using a `CommissionEmployee` Class

```
1  // Fig. 9.4: CommissionEmployee.java
2  // CommissionEmployee class represents an employee paid a
3  // percentage of gross sales.
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 percentage
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 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 CommissionEmployee constructor
```

Class CommissionEmployee extends class Object (from package java.lang).

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 6.)

- `CommissionEmployee` inherits `Object`'s methods.
- If you don't explicitly specify which class a new class extends, the class extends `Object` implicitly.


```
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first; // should validate
28    } // end method setFirstName
29
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; // should validate
40    } // end method setLastName
41
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 6.)

```
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
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 3 of 6.)


```
60     // set gross sales amount
61     public void setGrossSales( double sales )
62     {
63        if ( sales >= 0.0 )
64           grossSales = sales;
65        else
66           throw new IllegalArgumentException(
67              "Gross sales must be >= 0.0" );
68     } // end method setGrossSales
69
70     // return gross sales amount
71     public double getGrossSales()
72     {
73        return grossSales;
74     } // end method getGrossSales
75
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 4 of 6.)


```
76     // set commission rate
77     public void setCommissionRate( double rate )
78     {
79        if ( rate > 0.0 && rate < 1.0 )
80           commissionRate = rate;
81        else
82           throw new IllegalArgumentException(
83              "Commission rate must be > 0.0 and < 1.0" );
84     } // end method setCommissionRate
85
86     // return commission rate
87     public double getCommissionRate()
88     {
89        return commissionRate;
90     } // end method getCommissionRate
91
92     // calculate earnings
93     public double earnings()
94     {
95        return commissionRate * grossSales;
96     } // end method earnings
97
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 5 of 6.)


```
98     // return String representation of CommissionEmployee object
99     @Override // indicates that this method overrides a superclass method
100    public String toString()
101    {
102       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103          "commission employee", firstName, lastName,
104          "social security number", socialSecurityNumber,
105          "gross sales", grossSales,
106          "commission rate", commissionRate );
107    } // end method toString
108 } // end class CommissionEmployee
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 6 of 6.)

Creating and Using a CommissionEmployee Class (Cont.)

- Constructors are not inherited.
- The first task of a subclass constructor is to call its direct superclass's constructor explicitly or implicitly
  - Ensures that the instance variables inherited from the superclass are initialized properly.
- If the code does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default or no-argument constructor.
- A class's default constructor calls the superclass's default or no-argument constructor.

Creating and 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.

Creating and Using a CommissionEmployee Class (Cont.)

- To override a superclass method, a subclass must declare a method with the same signature as the superclass method
- `@Override` annotation
  - Indicates that a method should override a superclass method with the same signature.
  - If it does not, a compilation error occurs.

Common Programming Error 9.1

Using an incorrect method signature when attempting to override a superclass method causes an unintentional method overload that can lead to subtle logic errors.

Error-Prevention Tip 9.1

Declare overridden methods with the `@Override` *annotation to ensure at compilation time that you defined their signatures correctly. It's always better to find errors at compile time rather than at runtime.*

```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: \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() );
```

Fig. 9.5 | `CommissionEmployee` class test program. (Part 1 of 2.)


```java

      employee.setGrossSales( 500 ); // set gross sales
      employee.setCommissionRate( .1 ); // set commission rate

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

Implicit `toString` call occurs here

```
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
```

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

Case Study Part 2: Creating and Using a BasePlus-CommissionEmployee Class

- Class `BasePlusCommissionEmployee` contains a first name, last name, social security number, gross sales amount, commission rate and base salary.
  - All but the base salary are in common with class `CommissionEmployee`.
- Class `BasePlusCommissionEmployee`'s `public` services include a constructor, and methods `earnings`, `toString` and *get* and *set* for each instance variable
  - Most of these are in common with class `CommissionEmployee`.


```java
// Fig. 9.6: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class represents an employee who receives
// a base salary in addition to commission.

public class BasePlusCommissionEmployee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage
   private double baseSalary; // base salary per week

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

Class BasePlusCommissionEmployee does not specify "extends Object", Implicitly extends Object.

BasePlusCommissionEmployee's constructor invokes class Object's default constructor implicitly.

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 7.)

```
23          setCommissionRate( rate ); // validate and store commission rate
24          setBaseSalary( salary ); // validate and store base salary
25       } // end six-argument BasePlusCommissionEmployee constructor
26
27       // set first name
28       public void setFirstName( String first )
29       {
30          firstName = first; // should validate
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; // should validate
43       } // end method setLastName
44
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 7.)


```
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
57       // return social security number
58       public String getSocialSecurityNumber()
59       {
60          return socialSecurityNumber;
61       } // end method getSocialSecurityNumber
62
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 7.)


```
63       // set gross sales amount
64       public void setGrossSales( double sales )
65       {
66          if ( sales >= 0.0 )
67             grossSales = sales;
68          else
69             throw new IllegalArgumentException(
70                "Gross sales must be >= 0.0" );
71       } // end method setGrossSales
72
73       // return gross sales amount
74       public double getGrossSales()
75       {
76          return grossSales;
77       } // end method getGrossSales
78
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 4 of 7.)


```
79       // set commission rate
80       public void setCommissionRate( double rate )
81       {
82          if ( rate > 0.0 && rate < 1.0 )
83             commissionRate = rate;
84          else
85             throw new IllegalArgumentException(
86                "Commission rate must be > 0.0 and < 1.0" );
87       } // end method setCommissionRate
88
89       // return commission rate
90       public double getCommissionRate()
91       {
92          return commissionRate;
93       } // end method getCommissionRate
94
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 5 of 7.)

```
95      // set base salary
96      public void setBaseSalary( double salary )
97      {
98         if ( salary >= 0.0 )
99            baseSalary = salary;
100        else
101           throw new IllegalArgumentException(
102              "Base salary must be >= 0.0" );
103     } // end method setBaseSalary
104
105     // return base salary
106     public double getBaseSalary()
107     {
108        return baseSalary;
109     } // end method getBaseSalary
110
111     // calculate earnings
112     public double earnings()
113     {
114        return baseSalary + ( commissionRate * grossSales );
115     } // end method earnings
116
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 6 of 7.)


```
117     // return String representation of BasePlusCommissionEmployee
118     @Override // indicates that this method overrides a superclass method
119     public String toString()
120     {
121        return String.format(
122           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
123           "base-salaried commission employee", firstName, lastName,
124           "social security number", socialSecurityNumber,
125           "gross sales", grossSales, "commission rate", commissionRate,
126           "base salary", baseSalary );
127     } // end method toString
128  } // end class BasePlusCommissionEmployee
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 7 of 7.)


```
1   // Fig. 9.7: BasePlusCommissionEmployeeTest.java
2   // BasePlusCommissionEmployee test program.
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", "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() );
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 1 of 3.)


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

Case Study Part 2: Creating and Using a BasePlus-CommissionEmployee Class (Cont.)

- Much of `BasePlusCommissionEmployee`'s code is similar, or identical, to that of `CommissionEmployee`.
- `private` instance variables `firstName` and `lastName` and methods `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical.
  - Both classes also contain corresponding *get* and *set* methods.
- The constructors are almost identical
  - `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`.
- The `toString` methods are nearly identical
  - `BasePlusCommissionEmployee`'s `toString` also outputs instance variable `baseSalary`

Case Study Part 2: Creating and Using a BasePlus-CommissionEmployee Class (Cont.)

- We literally *copied* CommissionEmployee's code, pasted it into BasePlusCommissionEmployee, then modified the new class to include a base salary and methods that manipulate the base salary.
  - This "copy-and-paste" approach is often error prone and time consuming.
  - It spreads copies of the same code throughout a system, creating a **code-maintenance nightmare.**

Software Engineering Observation 9.3
With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are made for these common features 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.

Case Study Part 3: 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 the superclass's public and protected members are directly accessible in the subclass.

```
1  // Fig. 9.8: BasePlusCommissionEmployee.java
2  // private superclass members cannot be accessed in a subclass.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       // explicit call to superclass CommissionEmployee constructor
13       super( first, last, ssn, sales, rate );
14
15       setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 1 of 5.)


```
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       if ( salary >= 0.0 )
22          baseSalary = salary;
23       else
24          throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26    } // end method setBaseSalary
27
28    // return base salary
29    public double getBaseSalary()
30    {
31       return baseSalary;
32    } // end method getBaseSalary
33
34    // calculate earnings
35    @Override // indicates that this method overrides a superclass method
36    public double earnings()
37    {
38       // not allowed: commissionRate and grossSales private in superclass
39       return baseSalary + ( commissionRate * grossSales );
40    } // end method earnings
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 2 of 5.)


```
41
42    // return String representation of BasePlusCommissionEmployee
43    @Override // indicates that this method overrides a superclass method
44    public String toString()
45    {
46       // not allowed: attempts to access private superclass members
47       return String.format(
48          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
49          "base-salaried commission employee", firstName, lastName,
50          "social security number", socialSecurityNumber,
51          "gross sales", grossSales, "commission rate", commissionRate,
52          "base salary", baseSalary );
53    } // end method toString
54 } // end class BasePlusCommissionEmployee
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 3 of 5.)


```
BasePlusCommissionEmployee.java:39: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee.java:39: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee.java:49: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee.java:49: lastName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee.java:50: socialSecurityNumber has private access
in CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:51: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 4 of 5.)

```
BasePlusCommissionEmployee.java:51: commissionRate has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                       ^
7 errors
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 5 of 5.)

Case Study Part 3: Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)

- Each subclass constructor must implicitly or explicitly call its superclass constructor 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 subclass constructor's body.
- If the subclass constructor did not invoke the superclass's constructor explicitly, Java would attempt to invoke the superclass's no-argument or default 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.

Case Study Part 4: CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables

- To enable a subclass to directly access superclass instance variables, we can declare those members as protected in the superclass.
- New CommissionEmployee class modified only lines 6–10 of Fig. 9.4 as follows:

```
protected String firstName;
protected String lastName;
protected String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

- With protected instance variables, the subclass gets access to the instance variables, but classes that are not subclasses and classes that are not in the same package cannot access these variables directly.


```
1  // Fig. 9.9: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee inherits protected instance
3  // variables from CommissionEmployee.
4
5  public class BasePlusCommissionEmployee extends CommissionEmployee
6  {
7     private double baseSalary; // base salary per week
8
9     // six-argument constructor
10    public BasePlusCommissionEmployee( String first, String last,
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 BasePlusCommissionEmployee constructor
16
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 1 of 3.)

```
17     // set base salary
18     public void setBaseSalary( double salary )
19     {
20        if ( salary >= 0.0 )
21           baseSalary = salary;
22        else
23           throw new IllegalArgumentException(
24              "Base salary must be >= 0.0" );
25     } // end method setBaseSalary
26
27     // return base salary
28     public double getBaseSalary()
29     {
30        return baseSalary;
31     } // end method getBaseSalary
32
33     // calculate earnings
34     @Override // indicates that this method overrides a superclass method
35     public double earnings()
36     {
37        return baseSalary + ( commissionRate * grossSales );
38     } // end method earnings
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 2 of 3.)


```
39
40     // return String representation of BasePlusCommissionEmployee
41     @Override // indicates that this method overrides a superclass method
42     public String toString()
43     {
44        return String.format(
45           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
46           "base-salaried commission employee", firstName, lastName,
47           "social security number", socialSecurityNumber,
48           "gross sales", grossSales, "commission rate", commissionRate,
49           "base salary", baseSalary );
50     } // end method toString
51  } // end class BasePlusCommissionEmployee
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 3 of 3.)

Case Study Part 4: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Class `BasePlusCommissionEmployee` (Fig. 9.9) extends the new version of class `CommissionEmployee` with `protected` instance variables.
  - These variables are now `protected` members of `BasePlusCommissionEmployee`.
- If another class extends this version of class `BasePlusCommissionEmployee`, the new subclass also can access the `protected` members.
- The source code in Fig. 9.9 (51 lines) is considerably shorter than that in Fig. 9.6 (128 lines)
  - Most of the functionality is now inherited from `CommissionEmployee`
  - There is now only one copy of the functionality.
  - Code is easier to maintain, modify and debug—the code related to a commission employee exists only in class `CommissionEmployee`.

Case Study Part 4: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Inheriting **`protected`** instance variables slightly increases performance, because we can directly access the variables in the subclass without incurring the overhead of a *set or get method call.*
- In most cases, it's better to use **`private`** instance variables to encourage proper software engineering, and leave code optimization issues to the compiler.
  - Code will be easier to maintain, modify and debug.

Case Study Part 4: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Using `protected` instance variables creates several potential problems.
- The subclass object can set an inherited variable's value directly without using a *set method.*
  - A subclass object can assign an invalid value to the variable
- Subclass methods are more likely to be written so that they depend on the superclass's data implementation.
  - Subclasses should depend only on the superclass services and not on the superclass data implementation.
- We may need to modify all the subclasses of the superclass if the superclass implementation changes.
  - You should be able to change the superclass implementation while still providing the same services to the subclasses.

Error-Prevention Tip 9.2

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

Case Study Part 5: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables => BEST DESIGN

```
1  // Fig. 9.10: CommissionEmployee.java
2  // CommissionEmployee class uses methods to manipulate its
3  // private instance variables.
4  public class CommissionEmployee
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 CommissionEmployee( 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 CommissionEmployee constructor
```

instance variables are declared as private and public methods for manipulating these are provided.

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 1 of 6.)


```
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first; // should validate
28    } // end method setFirstName
29
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       la5stName = last; // should validate
40    } // end method setLastName
41
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 2 of 6.)

```
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
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 3 of 6.)


```
60     // set gross sales amount
61     public void setGrossSales( double sales )
62     {
63        if ( sales >= 0.0 )
64           grossSales = sales;
65        else
66           throw new IllegalArgumentException(
67              "Gross sales must be >= 0.0" );
68     } // end method setGrossSales
69
70     // return gross sales amount
71     public double getGrossSales()
72     {
73        return grossSales;
74     } // end method getGrossSales
75
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 4 of 6.)


```
76     // set commission rate
77     public void setCommissionRate( double rate )
78     {
79        if ( rate > 0.0 && rate < 1.0 )
80           commissionRate = rate;
81        else
82           throw new IllegalArgumentException(
83              "Commission rate must be > 0.0 and < 1.0" );
84     } // end method setCommissionRate
85
86     // return commission rate
87     public double getCommissionRate()
88     {
89        return commissionRate;
90     } // end method getCommissionRate
91
92     // calculate earnings
93     public double earnings()
94     {
95        return getCommissionRate() * getGrossSales();
96     } // end method earnings
97
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 5 of 6.)


```
98     // return String representation of CommissionEmployee object
99     @Override // indicates that this method overrides a superclass method
100    public String toString()
101    {
102       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103          "commission employee", getFirstName(), getLastName(),
104          "social security number", getSocialSecurityNumber(),
105          "gross sales", getGrossSales(),
106          "commission rate", getCommissionRate() );
107    } // end method toString
108 } // end class CommissionEmployee
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 6 of 6.)

Case Study Part 5: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- `CommissionEmployee` methods `earnings` and `toString` use the class's *get* methods to obtain the values of its instance variables.
  - If we decide to change the internal representation of the data (e.g., variable names) only the bodies of the *get and set methods that directly manipulate the instance variables will need to change.*
  - These changes occur solely within the superclass-—no changes to the subclass are needed.
  - Localizing the effects of changes like this is a good software engineering practice.
- Subclass `BasePlusCommissionEmployee` inherits `CommissionEmployee`'s non-private methods and can access the private superclass members via those methods.

Case Study Part 5: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

```
1  // Fig. 9.11: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class inherits from CommissionEmployee
3  // and accesses the superclass's private data via inherited
4  // public methods.
5
6  public class BasePlusCommissionEmployee extends CommissionEmployee
7  {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee( String first, String last,
12       String ssn, double sales, double rate, double salary )
13    {
14       super( first, last, ssn, sales, rate );
15       setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 1 of 3.)


```
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       if ( salary >= 0.0 )
22          baseSalary = salary;
23       else
24          throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26    } // end method setBaseSalary
27
28    // return base salary
29    public double getBaseSalary()
30    {
31       return baseSalary;
32    } // end method getBaseSalary
33
34    // calculate earnings
35    @Override // indicates that this method overrides a superclass method
36    public double earnings()
37    {
38       return getBaseSalary() + super.earnings();
39    } // end method earnings
```

Method `earnings` overrides class the superclass's `earnings` method.

calls `CommissionEmployee`'s `earnings` method with `super.earnings()`

Good software engineering practice: If a method performs all or some of the actions needed by another method, call that method rather than duplicate its code.


```
40
41    // return String representation of BasePlusCommissionEmployee
42    @Override // indicates that this method overrides a superclass method
43    public String toString()
44    {
45       return String.format( "%s %s\n%s: %.2f", "base-salaried",
46          super.toString(), "base salary", getBaseSalary() );
47    } // end method toString
48 } // 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.)

BasePlusCommissionEmployee's toString method overrides class CommissionEmployee's toString method

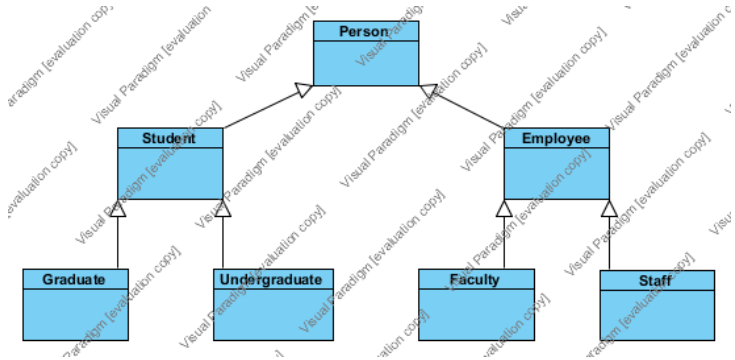
The new version creates part of the String representation by calling CommissionEmployee's toString method with the expression super.toString().

Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - The subclass constructor, before performing its own tasks, invokes its direct superclass's 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 class Object's constructor.
- Original subclass constructor's body finishes executing last.
- Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits.

UML Inheritance Diagrams

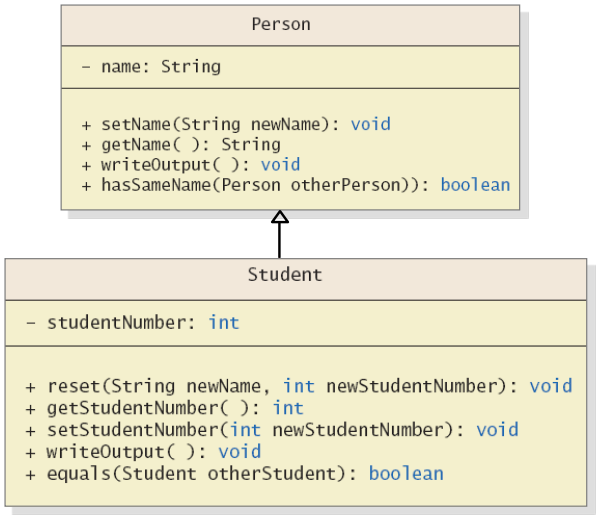
A class hierarchy in UML notation



An Employee is a Person and so forth; hence the arrows point up.

UML Inheritance Diagrams

- Some details of UML class hierarchy from previous figure



Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012