



Chapter 10: Inheritance

Yepang LIU

liuyp1@sustech.edu.cn



Objectives

- ▶ Inheritance (继承)
- ▶ Superclass (父类) and subclass (子类)
- ▶ Overriding (重写)

Inheritance

- ▶ Consider a scenario where you have carefully designed and implemented a **Vehicle** class, and you need a **Truck** class in your system. Will you create the new class from scratch?

On one hand, trucks have some traits in common with many vehicles. Some code can be shared (So no?)



On another hand, trucks have their own characteristics e.g., two seats, can carry huge things (So yes?)

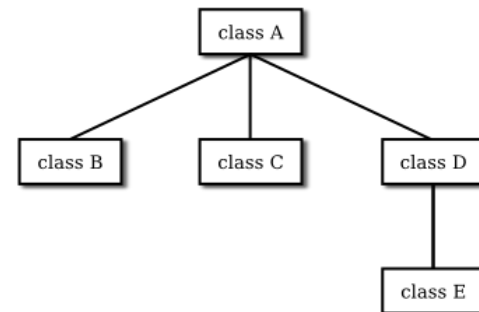


Inheritance

- ▶ A form of software reuse in which a new class is created by absorbing (吸收复用) an existing class's members and embellishing (修饰) them with new or modified capabilities.
- ▶ Can save time during program development by basing new classes on existing high-quality software.
- ▶ Increases the likelihood that a system can 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**
- ▶ Each subclass can be a superclass of future subclasses, forming a **class hierarchy**.



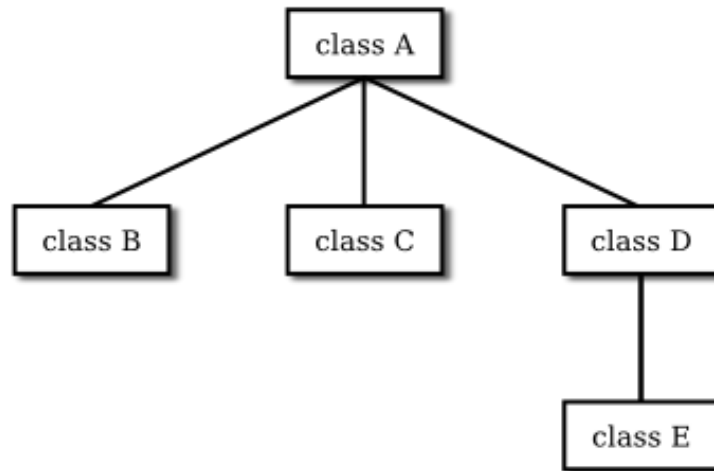


Inheritance

- ▶ A subclass can add its own fields and methods.
- ▶ A subclass is **more specific** than its superclass and represents a more specialized group of objects.
- ▶ 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**.

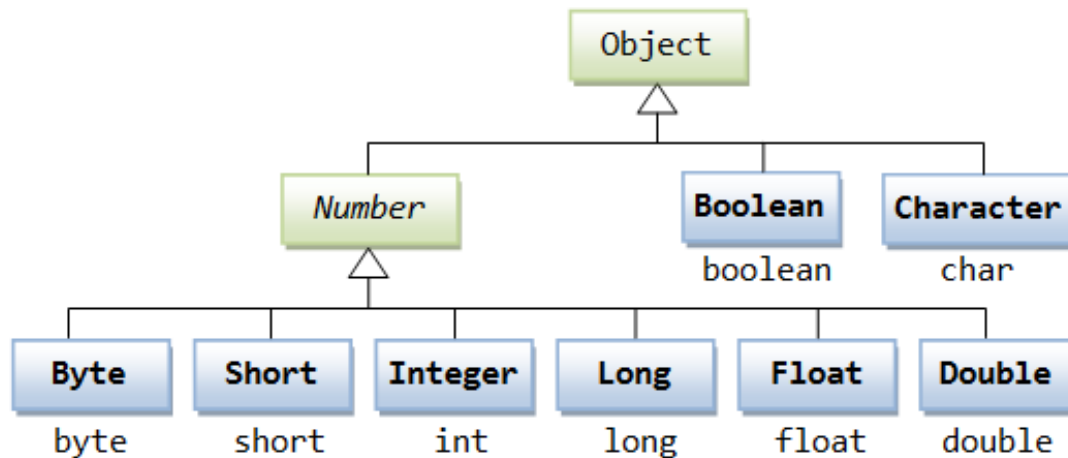
Inheritance

- ▶ The **direct superclass** is the superclass from which the subclass explicitly inherits (A is the direct superclass of C)
- ▶ An **indirect superclass** is any class above the direct superclass in the **class hierarchy** (e.g., A is an indirect superclass of E)



Inheritance

- ▶ The Java class hierarchy begins with class `java.lang.Object`
 - *Every* class directly or indirectly **extends** (or “inherits from”) `Object`.
- ▶ Java supports only **single inheritance**, in which each class is derived from exactly one direct superclass.



Inheritance vs. Composition

- ▶ **Inheritance:** *Is-a* relationship between classes
 - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass (a truck is also a vehicle)
- ▶ **Composition:** *Has-a* relationship between classes
 - In a *has-a* relationship, an object contains references to other objects as members (a house contains a kitchen)

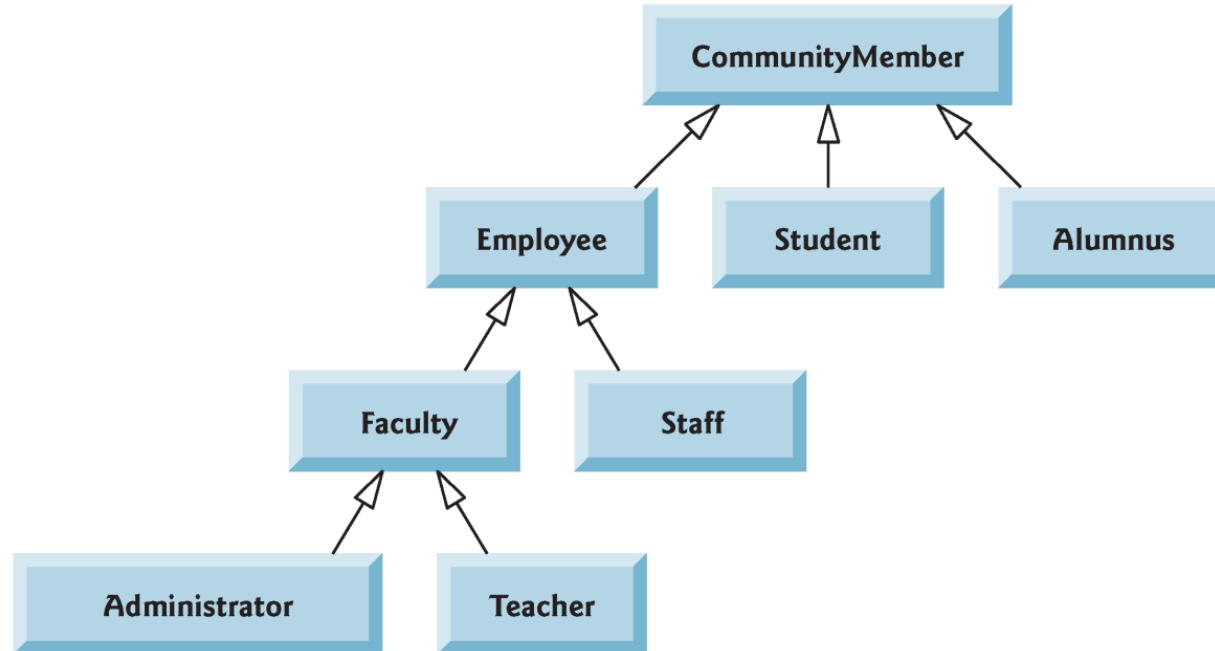
More Examples and Observations

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

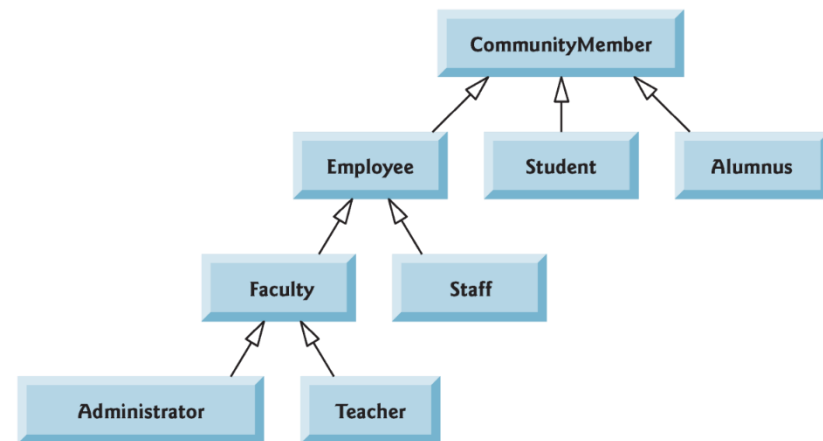
- ▶ Superclasses tend to be “more general” and subclasses “more specific”
- ▶ Because every subclass object *is an object of its superclass*, and one superclass can have many subclasses, the set of objects represented by a superclass is typically larger than the set of objects represented by any of its subclasses.

Class Hierarchy (Inheritance Hierarchy)

- ▶ A superclass exists in a hierarchical relationship with its subclasses.



Class Hierarchy



- ▶ Each arrow in the hierarchy represents an *is-a relationship*.
- ▶ Follow the arrows upward in the class hierarchy
 - a Teacher *is a* Faculty (also an Employee, a CommunityMember)
- ▶ CommunityMember is the direct superclass of Employee, Student and Alumnus and is an indirect superclass of all the other classes



Superclass and Subclass (Cont.)

- ▶ Objects of all classes that extend a common superclass can be treated as objects of that superclass (e.g., `java.lang.Object`)
 - Commonality expressed in the members of the superclass.
- ▶ Inheritance issue
 - 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 (there will be an example later)



public and private 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 (invisible to subclasses).



protected Members

- ▶ `protected` access is an intermediate access level between `public` and `private`
- ▶ A superclass's `protected` members can be accessed by
 - members of that superclass
 - members of its subclasses
 - 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.

protected Members

- ▶ 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.

Access Level Modifiers (ALL)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N*	N
private	Y	N	N	N

Note that this is for controlling access to class members. At the top level, a class can only be declared as `public` or package-private (no explicit modifier)

* Package-private class members are inherited if the subclass is in the same package as the superclass



Case Study: A Payroll Application

(工资表应用程序)

- ▶ Suppose we need to create classes for two types of employees
 - **Commission employees (佣金员工)** are paid a percentage of their sales (ComissionEmployee)
 - **Base-salaried commission employees (持底薪佣金员工)** receive a base salary plus a percentage of their sales (BasePlusCommissionEmployee)



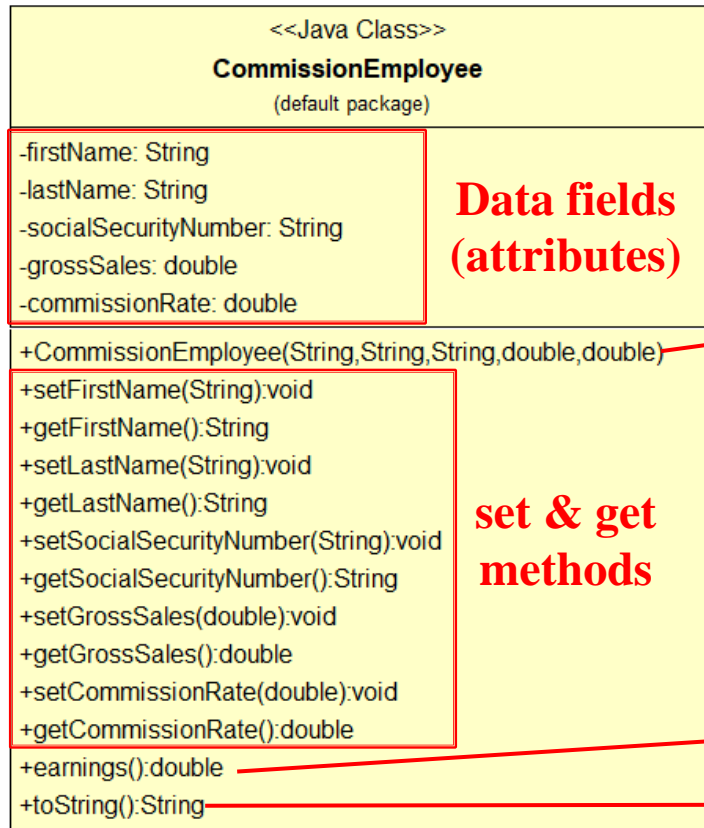
Comparing the Two Types

- ▶ Both classes need data fields to store the employee's personal information (e.g., name, social security number)

- ▶ **Differences**
 - BasePlusCommissionEmployee class needs one additional field to store the employee's base salary
 - The way of calculating the earnings are different

Design Choice #1

- ▶ Creating the two classes independently



**Data fields
(attributes)**

A five-argument constructor

**set & get
methods**

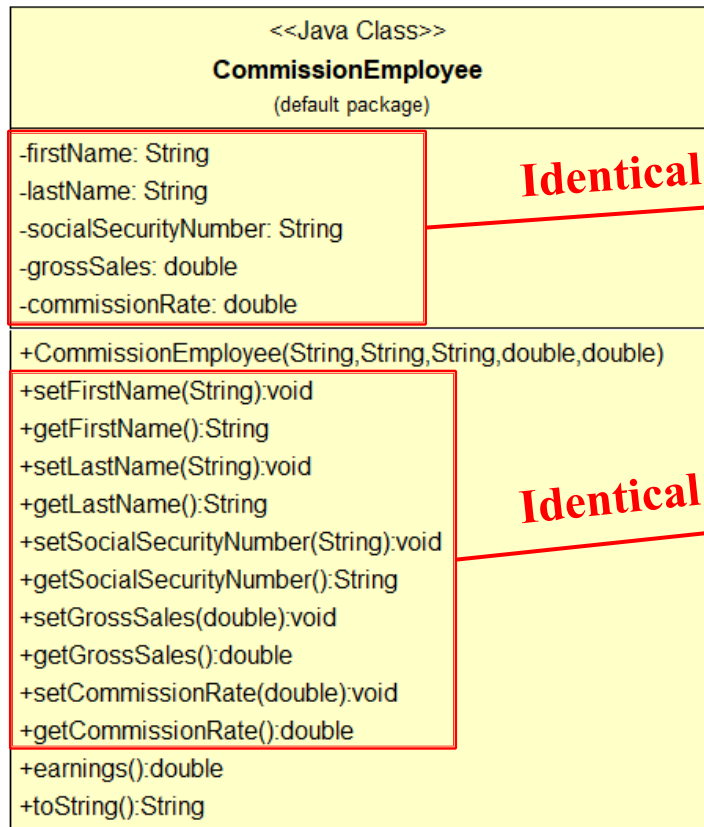
Earning calculation method

Transform object to string representation

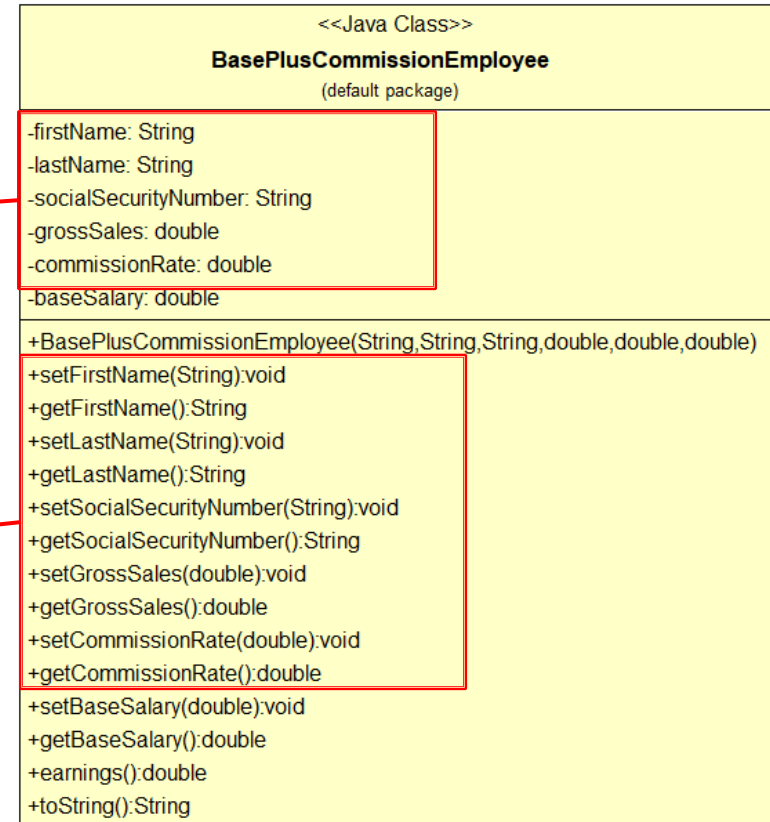
extends Object

Design Choice #1

- ▶ Creating the two classes independently



extends Object



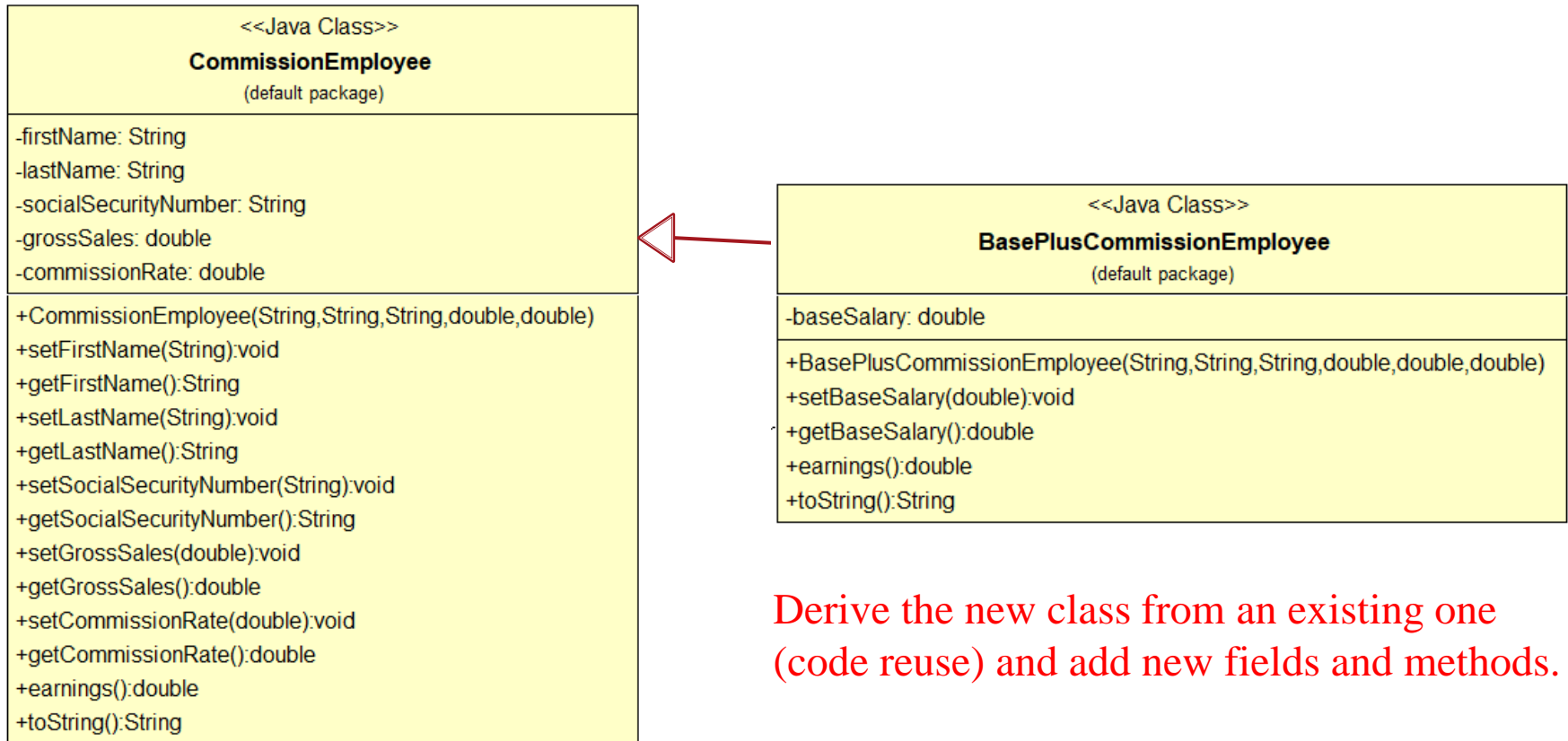
extends Object

Problems of Design Choice #1

- ▶ Much of `BasePlusCommissionEmployee`'s code will be identical to that of `CommissionEmployee`
- ▶ For implementation, we will literally copy the code of the class `CommissionEmployee` and paste the copied code into the class `BasePlusCommissionEmployee`
 - “Copy-and-paste” approach is often error prone. It spreads copies of the same code throughout a system, creating code-maintenance nightmares

Design Choice #2

- ▶ BasePlusCommissionEmployee extends CommissionEmployee



extends Object

Derive the new class from an existing one (code reuse) and add new fields and methods.



Let's get to the real code and use it to
illustrate some key notions



CommissionEmployee

```
public class CommissionEmployee extends Object {  
    private String firstName;  
    private String lastName;  
    private String socialSecurityNumber;  
    private double grossSales;  
    private double commissionRate;  
    ...  
}
```

“**extends Object**” is optional. If you don’t explicitly specify which class a new class extends, the class extends **Object** implicitly.

CommissionEmployee

A five-argument constructor

```
public CommissionEmployee(String first, String last, String ssn,  
                           double sales, double rate) {  
    firstName = first;  
    lastName = last;  
    socialSecurityNumber = ssn;  
    setGrossSales(sales); // data validation  
    setCommissionRate(rate); // data validation  
}  
  
public void setGrossSales(double sales) {  
    grossSales = (sales < 0.0) ? 0.0 : sales;  
}  
  
public void setCommissionRate(double rate) {  
    commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;  
}
```



CommissionEmployee

Several other get and set methods

```
public void setFirstName(String first) { firstName = first; }
public String getFirstName() { return firstName; }
public void setLastName(String last) { lastName = last; }
public String getLastName() { return lastName; }
public void setSocialSecurityNumber(String ssn) { socialSecurityNumber = ssn; }
public String getSocialSecurityNumber() { return socialSecurityNumber; }
public double getGrossSales() { return grossSales; }
public double getCommissionRate() { return commissionRate; }
```



CommissionEmployee

Calculation and string transformation methods

```
public double earnings() {  
    return commissionRate * grossSales;  
}
```

```
@Override  
public String toString() {  
    return String.format("%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",  
        "commission employee", firstName, lastName,  
        "social security number", socialSecurityNumber,  
        "gross sales", grossSales,  
        "commission rate", commissionRate);  
}
```

Overriding toString() Method

- ▶ `toString()` is one of the methods that every class inherits directly or indirectly from class `Object`.
 - Returns a `String` that “textually represents” an object.
 - Called implicitly whenever an object must be converted to a `String` representation (e.g., `System.out.println(objRef)`)
- ▶ Class `Object`’s `toString()` method returns a `String` that includes the name of the object’s class.
 - If not overridden, returns something like “`CommissionEmployee@70dea4e`” (the part after `@` is the hexadecimal representation of the hash code of the object)
 - This is primarily a placeholder that can be overridden by a subclass to specify customized `String` representation.



Overriding toString() Method

- ▶ To override a superclass's method, a subclass must declare a method with the **same signature** as the superclass method.
- ▶ The access level of an overriding method can be higher, but not lower than that of the overridden method (package-private < protected < public)
- ▶ **@Override annotation**
 - Optional, but helps the compiler to ensure that the method has the same signature as the one in the superclass

```
@Override  
public String toString() { ... }
```

BasePlusCommissionEmployee



```
public class BasePlusCommissionEmployee extends CommissionEmployee {  
    private double baseSalary; ← Added a new field
```

```
    public BasePlusCommissionEmployee(String first, String last, String ssn,  
                                     double sales, double rate, double salary) {  
        super(first, last, ssn, sales, rate);  
        setBaseSalary(salary);  
    }
```

Its own constructor. In Java, constructors are not class members and are not inherited by subclasses

```
    public void setBaseSalary(double salary) {  
        baseSalary = (salary < 0.0) ? 0.0 : salary;  
    }  
    ...
```

Subclass Constructors

- ▶ Each constructor in the subclass needs to invoke a superclass constructor for object construction (e.g., to initialize inherited instance variables) by using the **super** keyword.
- ▶ If this is not explicitly done, the compiler automatically inserts a call to the no-argument constructor of the superclass. **If the super class does not have a no-argument constructor, you will get a compile-time error.**
- ▶ **Constructor chaining:** If a subclass constructor invokes a constructor of its superclass, there will be a whole chain of constructors called, all the way back to the constructor of Object. You need to be aware of its effect.



BasePlusCommissionEmployee

Overriding the two inherited methods for customized behaviors

```
public double earnings() {  
    return baseSalary + ( commissionRate * grossSales );  
}
```

```
@Override  
public String toString() {  
    return String.format("%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",  
        "base-salaried", firstName, lastName,  
        "social security number", socialSecurityNumber,  
        "gross sales", grossSales,  
        "commission rate", commissionRate,  
        "base salary", baseSalary);  
}
```

Accessing fields of superclass

- ▶ What would happen if we compile the previous code?

Compilation error

```
BasePlusCommissionEmployee.java:35: commissionRate has private access in  
CommissionEmployee
```

```
    return baseSalary + ( commissionRate * grossSales );  
                           ^
```

```
BasePlusCommissionEmployee.java:35: grossSales has private access in  
CommissionEmployee
```

```
    return baseSalary + ( commissionRate * grossSales );  
                                   ^
```

...

```
public double earnings() {  
    return baseSalary + ( commissionRate * grossSales );  
}
```



Accessing fields of superclass

- ▶ A subclass inherits all public and protected members of its parent, no matter what package the subclass is in.
 - These members are directly accessible in the subclass
- ▶ If the subclass is in the same package as its parent, it also inherits the parent's *package-private* members (those without access level modifiers)
 - These members are directly accessible in the subclass
- ▶ A subclass does not inherit the private members. Private fields need to be accessed using the methods (public, protected, or package-private ones) inherited from superclass.

Solving the Compilation Problem

- ▶ **Solution #1:** using inherited methods

```
public double earnings() {  
    return baseSalary + ( commissionRate * grossSales );  
}
```

```
public double earnings() {  
    return baseSalary + ( getCommissionRate() * getGrossSales() );  
}
```



Solving the Compilation Problem

- **Solution #2:** declaring superclass fields as protected

```
public class CommissionEmployee extends Object {  
    private protected String firstName;  
    private protected String lastName;  
    private protected String socialSecurityNumber;  
    private protected double grossSales;  
    private protected double commissionRate;  
    ...  
}
```

Recall the Access Level Modifiers

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

Note that this is for controlling access to class members. At the top level, a class can only be declared as `public` or package-private (no explicit modifier)

Comparing the Solutions

- ▶ Inheriting protected instance variables (solution #2) slightly increases performance, because we directly access the variables in the subclass without incurring the overhead of set/get method calls.

```
public double earnings() {  
    return baseSalary + ( commissionRate * grossSales );  
}
```

VS.

```
public double earnings() {  
    return baseSalary + ( getCommissionRate() * getGrossSales() );  
}
```



Problems of protected Members

- ▶ **(Problem #1)** The subclass object can set an inherited protected variable's value directly without using a set method. **The value could be invalid**, leaving the object in an inconsistent state.
- ▶ **(Problem #2)** If protected variables are used in many methods in the subclass, these methods will depend on the superclass's data implementation
 - **Subclasses should depend only on the superclass services (e.g., public methods) and not on the superclass data implementation**
- ▶ **(Problem #3)** A class's **protected** members are visible to all classes in the same package, this is not always desirable.

Comparing the Solutions

- ▶ From the point of good software engineering, it's better to use private instance variables (solution #1) and leave code optimization issues to the compiler.
 - Code will be easier to maintain, modify and debug.

```
public double earnings() {  
    return baseSalary + ( getCommissionRate() * getGrossSales() );  
}
```

VS.

```
public double earnings() {  
    return baseSalary + ( commissionRate * grossSales );  
}
```



More on the **super** Keyword

- ▶ Two main usage scenarios
- ▶ The **super** keyword can be used to invoke a superclass's constructor (as illustrated by our earlier example)
- ▶ If your method overrides its superclass's method, you can invoke the overridden method using the keyword **super**.

More on the **super** Keyword

```
public class BasePlusCommissionEmployee extends CommissionEmployee {
    @Override
    public String toString() {
        return String.format("%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
            "base-salaried", getFirstName(), getLastName(),
            "social security number", getSocialSecurityNumber(),
            "gross sales", getGrossSales(),
            "commission rate", getCommissionRate(),
            "base salary", getBaseSalary());
    }
}

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



Using CommissionEmployee

```
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

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: 500.00
commission rate: 0.10

Using 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( "\n%s:\n\n%s\n",
    "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



Inheritance in a Nut Shell

- ▶ **The idea of inheritance is simple but powerful:** When you want to create a new class and there is already a class that includes some code you want, you can derive the new class from the existing one.
- ▶ The new class inherits its superclass's members—though the private superclass members are hidden in the subclass.



Inheritance in a Nut Shell

- ▶ You can **customize** the new class to meet your needs by including **additional members** and by **overriding superclass members**.
 - This does not require the subclass programmer to change (or even have access to) the superclass's source code.
 - Java simply requires access to the superclass's `.class` file.
- ▶ By doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.