Chapter 9: Inheritance

Yepang LIU (刘烨庞)

liuyp1@sustech.edu.cn

Objectives

- Inheritance (继承)
- Superclass (父类) and subclass (子类)
- The `protected` keyword
- Method overriding (重写)

A Motivating Example

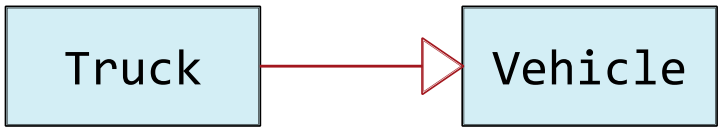
- Consider a scenario where you have carefully designed and implemented a `Vehicle` class, and you need a `Truck` class in your system. Will you create the new class from scratch?

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?)

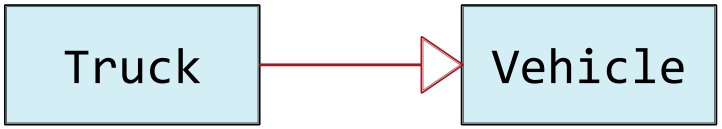
Inheritance (继承)

- Rather than declaring completely new members, you can designate that the new class should inherit (reuse) the members of an existing class.
  - Vehicle class is the superclass (base class, parent class)
  - Truck class is the subclass (derived class, child class)



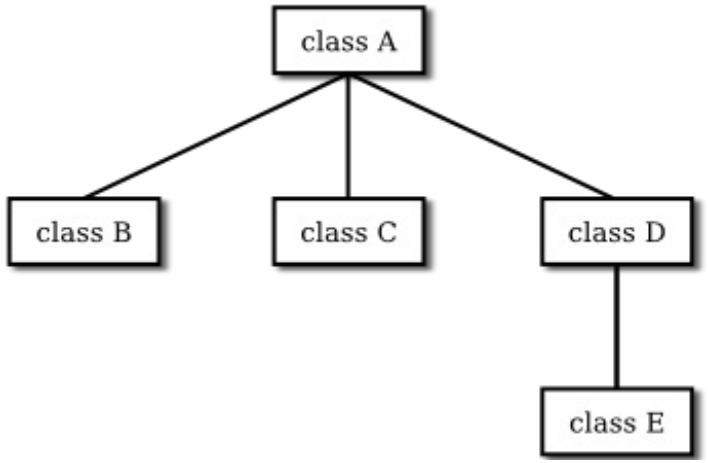
Superclass & Subclass

- A subclass inherits the fields and methods of its superclass
- A subclass can add its own fields and methods.
- Reusability: The subclass exhibits/reuses the behaviors of its superclass and can add new behaviors that are specific to the subclass.
  - This is why inheritance is sometimes referred to as specialization (特殊化).



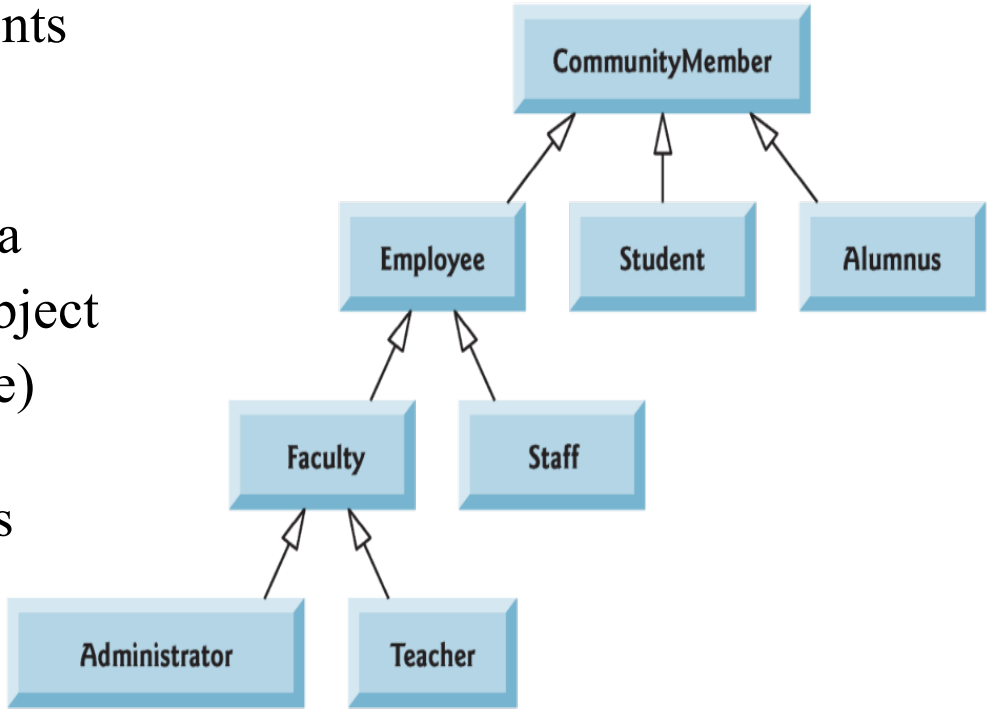
Class Hierarchy

- Each subclass can be a superclass of future subclasses, forming a class hierarchy (类层次结构)
- 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)



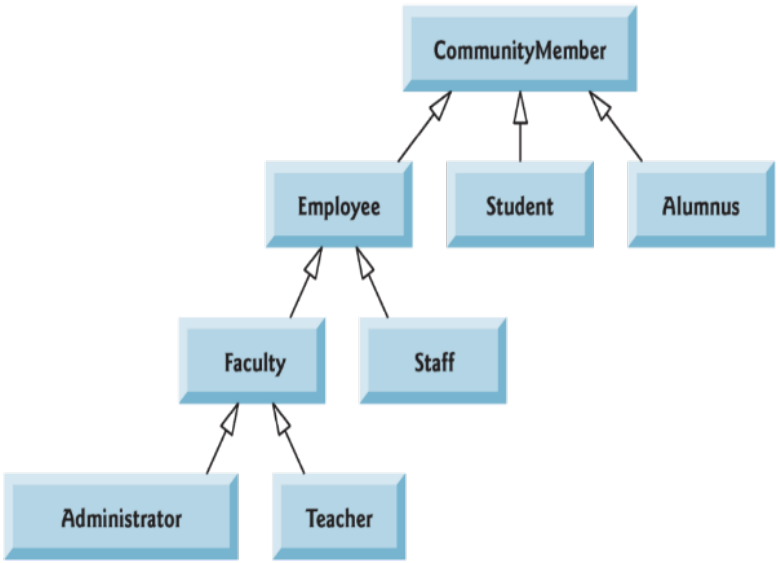
Class Hierarchy

- Each arrow in the hierarchy represents an *is-a relationship*.
- In an is-a relationship, an object of a subclass can also be treated as an object of its superclass (a truck **is a** vehicle)
- Follow the arrows upward in the class hierarchy
  - a `Teacher` **is a** `Faculty` (also an `Employee`, a `CommunityMember`)

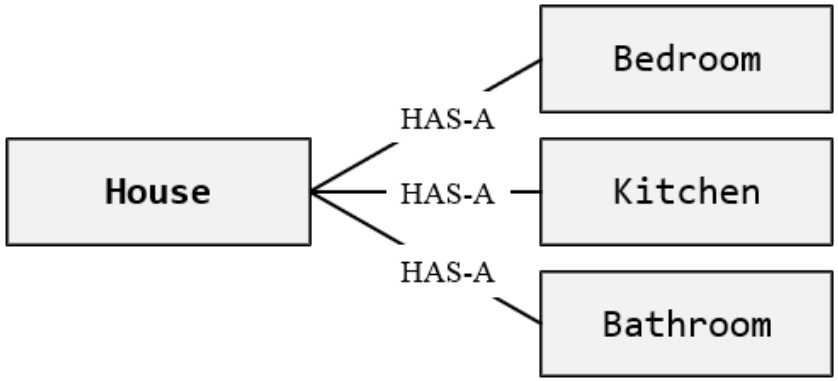


Inheritance vs. Composition

- **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)

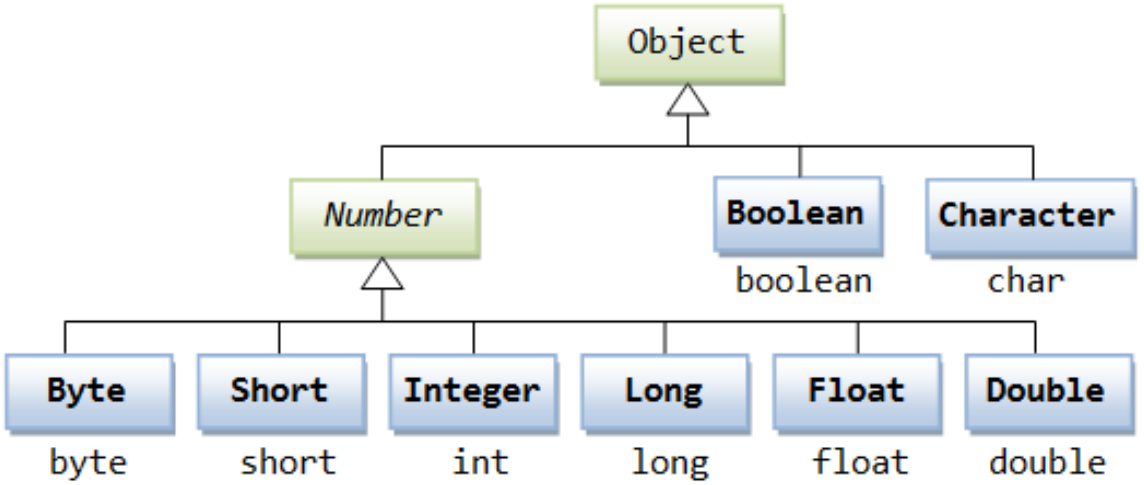


- **Composition:** *Has-a* relationship between classes
  - In a *has-a* relationship, an object contains references to other objects as members (a house contains a kitchen)



`Object`: the Cosmic Superclass

- 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.



More Examples

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

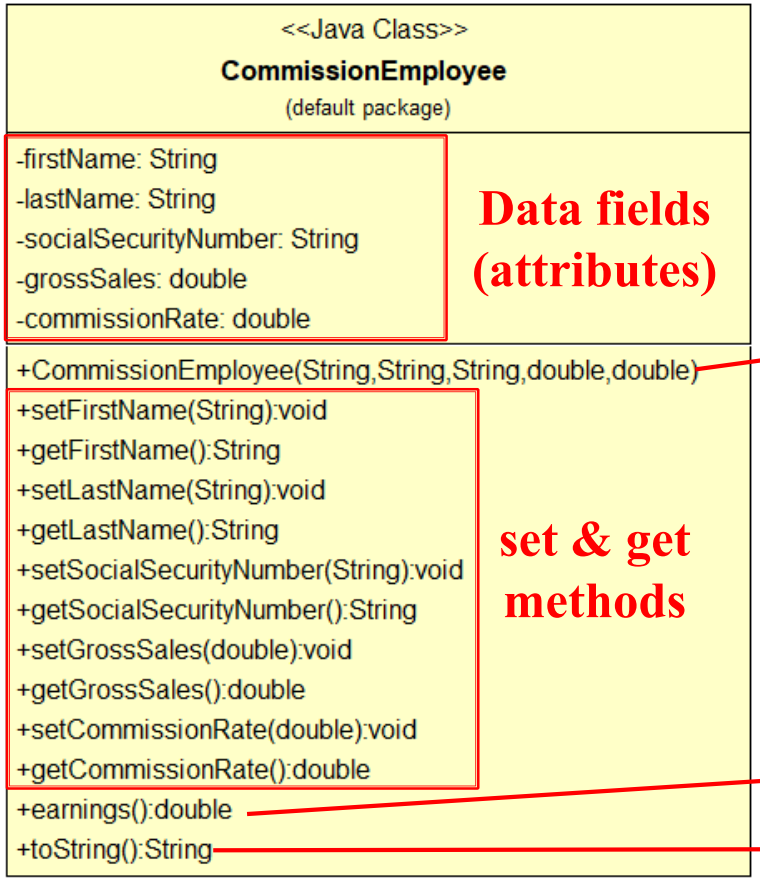
Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

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`)

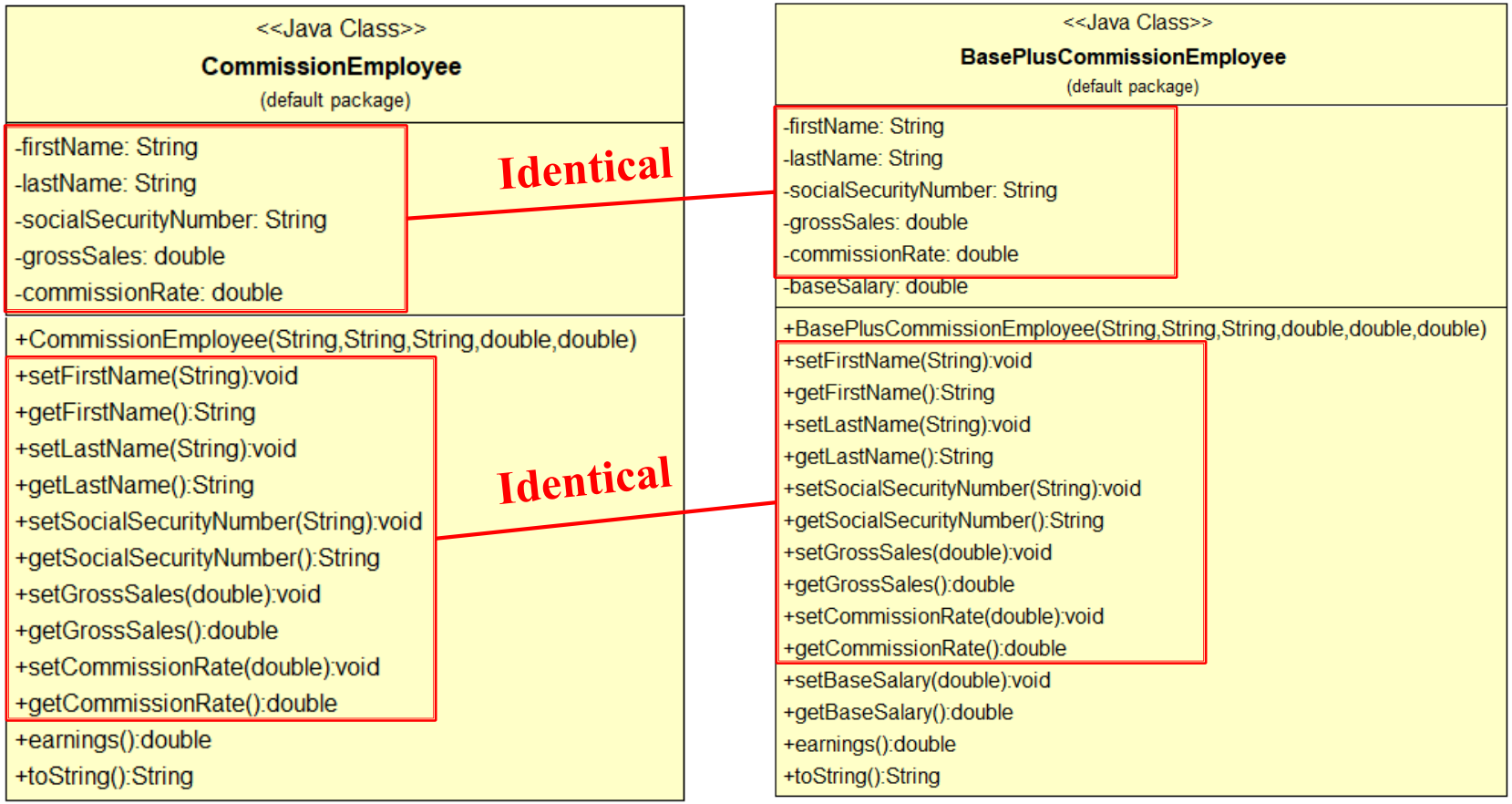
Design Choice #1

- Creating the two classes independently



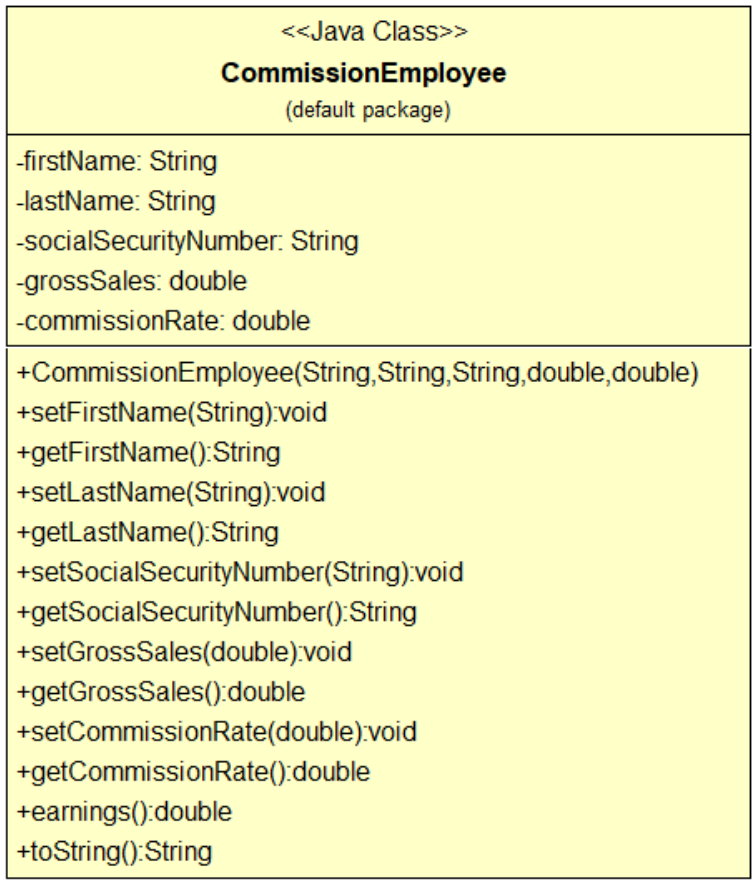
Design Choice #1

- Creating the two classes independently

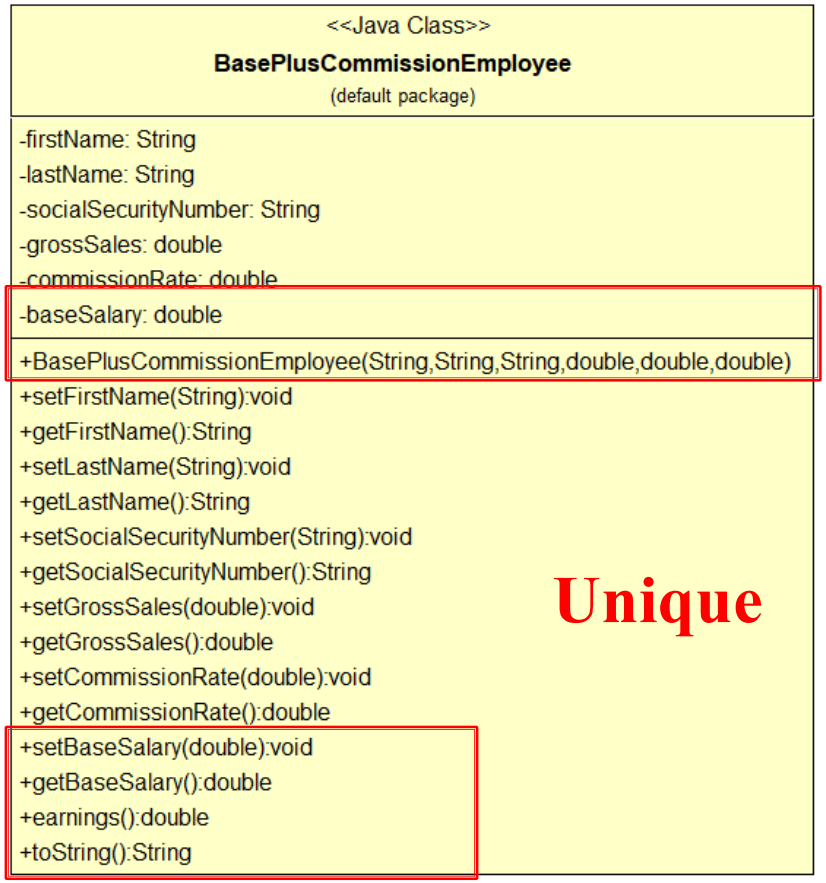


Design Choice #1

- Creating the two classes independently



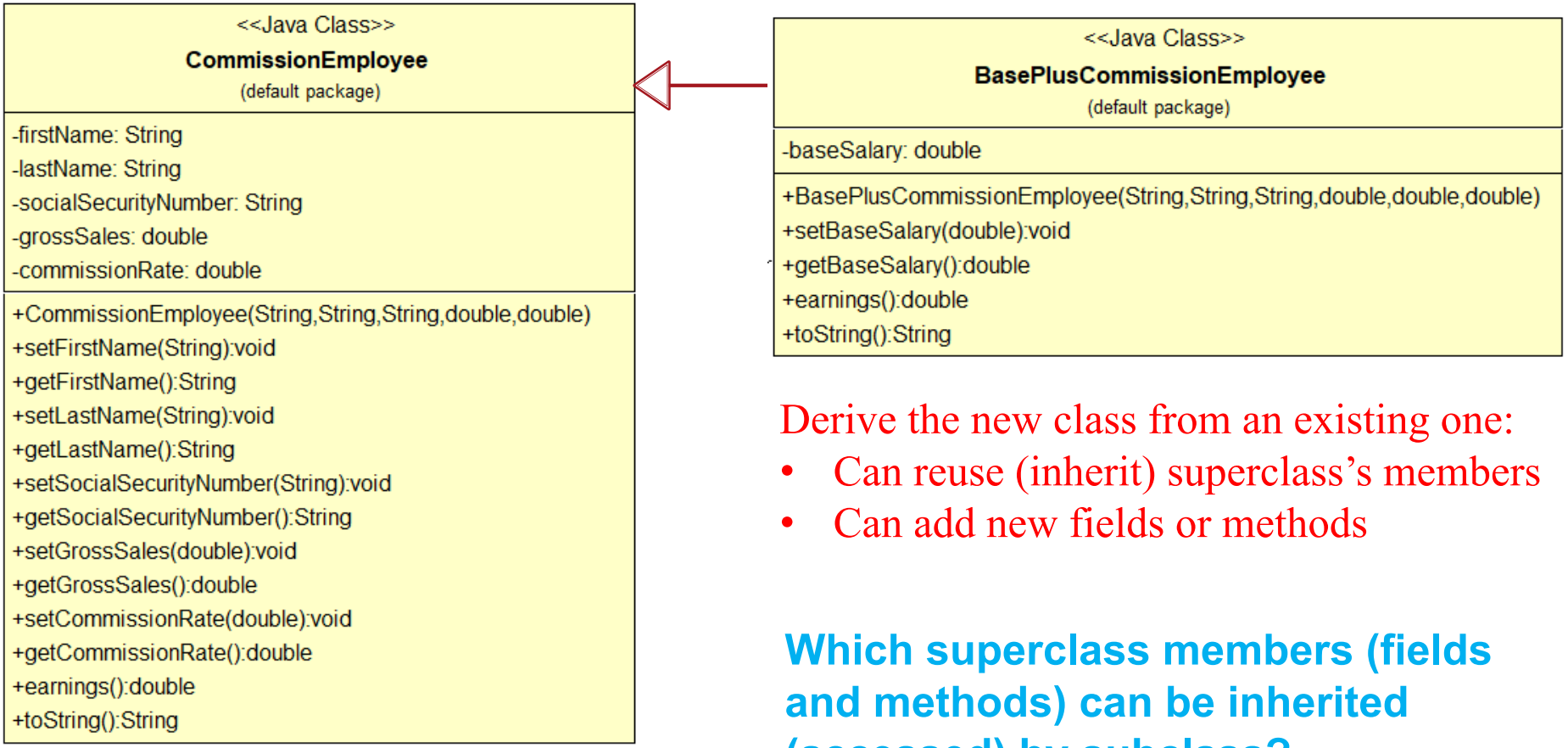
extends `Object`



extends `Object`

Design Choice #2

- `BasePlusCommissionEmployee` extends `CommissionEmployee`



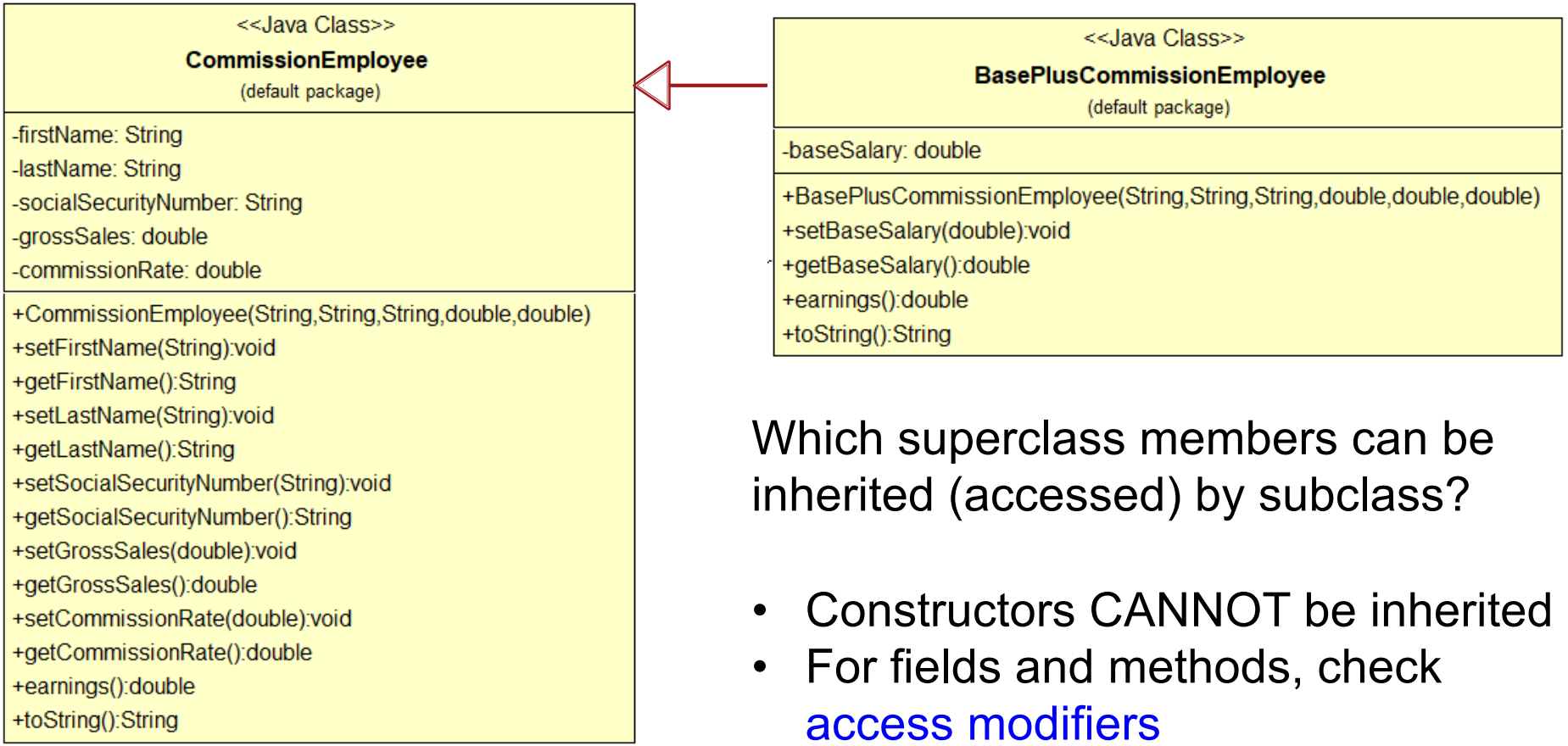
Derive the new class from an existing one:

- Can reuse (inherit) superclass's members
- Can add new fields or methods

Which superclass members (fields and methods) can be inherited (accessed) by subclass?

Design Choice #2

- `BasePlusCommissionEmployee` extends `CommissionEmployee`



Which superclass members can be inherited (accessed) by subclass?

- Constructors CANNOT be inherited
- For fields and methods, check access modifiers

Access Level Modifiers (ALL)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N*	N
private	Y	N	N	N

A subclass inherits all public members of its superclass, no matter what package the subclass is in.
Public members in the superclass are directly accessible in the subclass

Access Level Modifiers (ALL)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N*	N
private	Y	N	N	N

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

Access Level Modifiers (ALL)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N*	N
private	Y	N	N	N

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

Access Level Modifiers (ALL)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N*	N
private	Y	N	N	N

A subclass does NOT inherit the private members, or, private fields are hidden from the subclass

Private fields need to be accessed using the public, protected, or package-private methods inherited from superclass.

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

```
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;
}
```

A five-argument constructor

CommissionEmployee

Several other getter and setter 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);
}
```

BasePlusCommissionEmployee

Declare the superclass

```
public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary;
```

Add 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);
    }
```

Subclass's 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;
    }
    ...
```

BasePlusCommissionEmployee

```
public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary;

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

- The super keyword can be used to invoke a superclass's constructor
- Invocation of a superclass constructor must be the first line in the subclass constructor. This ensures that properties inherited from the superclass are set up first correctly.

BasePlusCommissionEmployee

```
public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary;

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

Compiler inserts `super();`

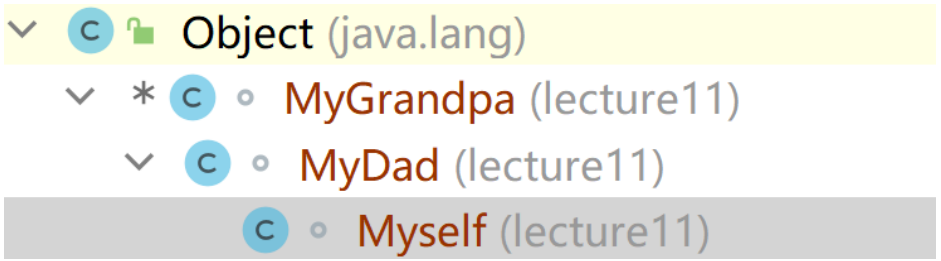
If super is not explicitly invoked, the compiler automatically inserts a call to the no-argument constructor of the superclass.

If the super class does not have a no-argument constructor, you will get a compile-time error.

Constructor Chaining

```
class MyGrandpa {
    1 usage
    MyGrandpa(){
        System.out.println("Grandpa");
    }
}

1 usage    1 inheritor
class MyDad extends MyGrandpa {
    1 usage
    MyDad(String name){
        System.out.println("Dad " + name);
    }
}

2 usages
class Myself extends MyDad {
    1 usage
    Myself(){
        super( name: "Joe"); // comment this?
        System.out.println("Me");
    }
}
```

```
public static void main(String[] args) {
    Myself me = new Myself();
}
```




```
Grandpa
Dad Joe
Me
```

BasePlusCommissionEmployee

Overriding (重写) the two inherited methods for customized behaviors

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

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);
}
```

BasePlusCommissionEmployee

Overriding (重写) the two inherited methods for customized behaviors

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

```
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 );
                                             ^
```

```
}
```

Compilation error!

BasePlusCommissionEmployee

Overriding (重写) the two inherited methods for customized behaviors

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

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);
}
```

Compilation error!

Solving the Compilation Problem

- Design #1: using inherited methods

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

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

Solving the Compilation Problem

- Design #2: declaring superclass fields as `protected`

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

Subclass methods can refer to `public` and `protected` members inherited from the superclass simply by using the member names.

Comparing the Designs

- 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 Designs

- From the point of data encapsulation, 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 );
}
```

The super Keyword: usage #2

```
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());

    }
}
```

What if we don't use `super.`?

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() );
```

Invoke subclass's constructor, which invokes superclass's constructor first

Inherited from superclass, and can be invoked directly because public

Inherited from superclass

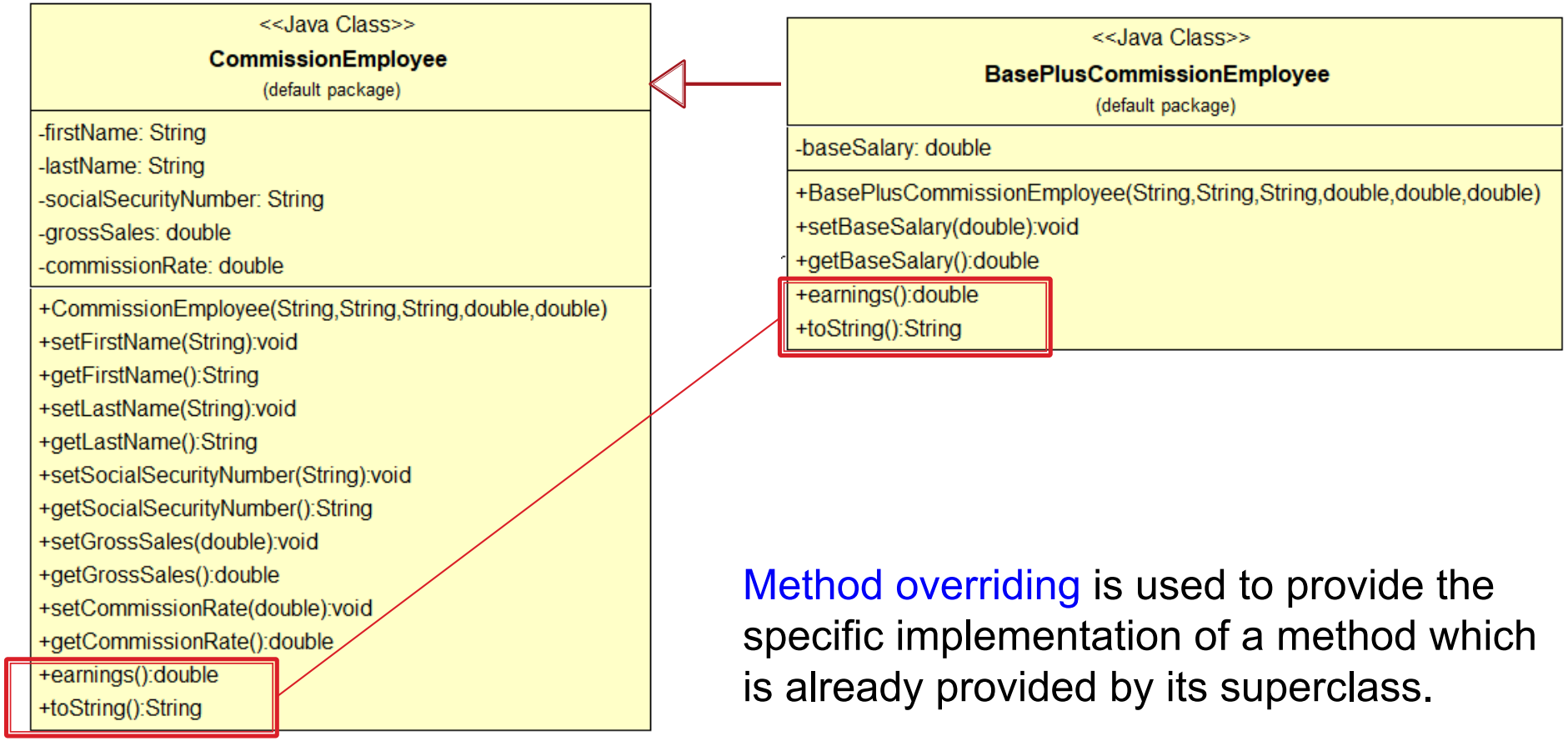
Defined in subclass

```
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
```

Method overriden in subclass

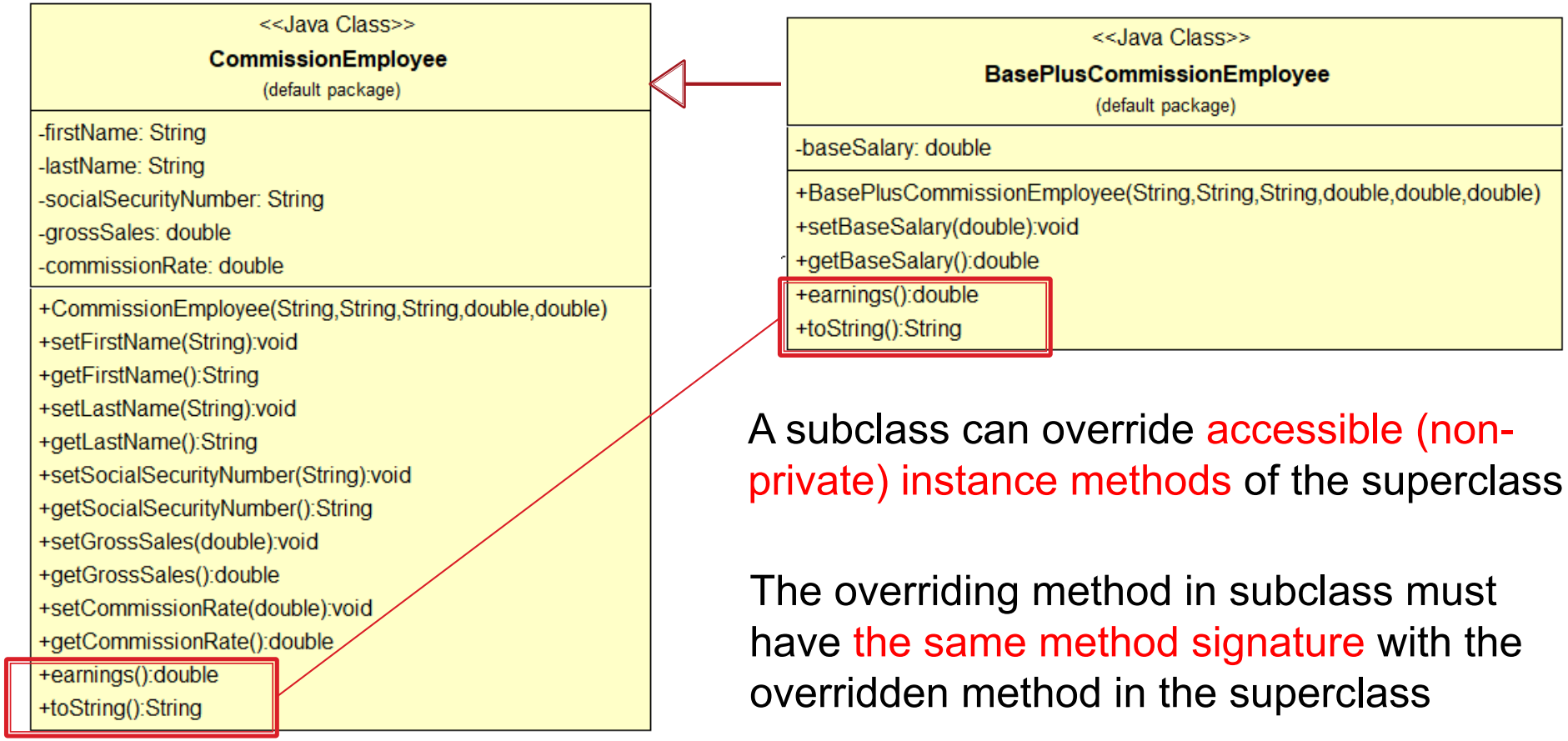
```
Employee information obtained by get methods:

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
```

Method Overriding (方法重写)



Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

Method Overriding (方法重写)



A subclass can override accessible (non-private) instance methods of the superclass

The overriding method in subclass must have the same method signature with the overridden method in the superclass

Overriding earnings()

- In superclass `CommissionEmployee`

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

- In subclass `BasePlusCommissionEmployee`

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

Overriding `toString()`

- In superclass `CommissionEmployee`

```
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);
}
```

- In subclass `BasePlusCommissionEmployee`

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

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 `equals(Object)` Method

- equals(Object) is another method that every class inherits directly or indirectly from class `Object`.

```
public boolean equals(Object o)
```

- This method checks whether two objects are equal.

```
object1.equals(object2)
```

- By default, the method tests whether two reference variables refer to the same object.

```
public boolean equals(Object obj){
        return this == obj;
}
```

Overriding `equals(Object)` Method

- We often want to implement state-based equality testing, in which two objects are considered equal when they have the same state (e.g., id). For this purpose, we have to override the equals method.

```
public boolean equals(Object otherObject){
    if(this == otherObject) return true;
    if(otherObject == null) return false;
    if(!(otherObject instanceof CommissionEmployee)){
        return false;
    }
    // type cast
    CommissionEmployee other = (CommissionEmployee) otherObject;
    return Objects.equals(socialSecurityNumber, other.socialSecurityNumber);
}
```

`Objects.equals(a, b)` returns true if both arguments are null, false if only one is null, and calls `a.equals(b)` otherwise.

Method Overriding

```
class Grandpa {
    public void hi()
    {
        System.out.println("I'm grandpa");
    }
}

class Dad extends Grandpa {
    public void hi()
    {
        System.out.println("I'm dad");
    }
}

class Me extends Dad {

    public void hi(String name)
    {
        System.out.println("I'm " + name);
    }
}
```

```
public class OverrideDemo {
    public static void main(String[] args) {
        Grandpa g = new Grandpa();
        Dad d = new Dad();
        Me m = new Me();

        g.hi();
        d.hi();
        m.hi();
        m.hi( name: "Joe");
    }
}
```

What's the output?

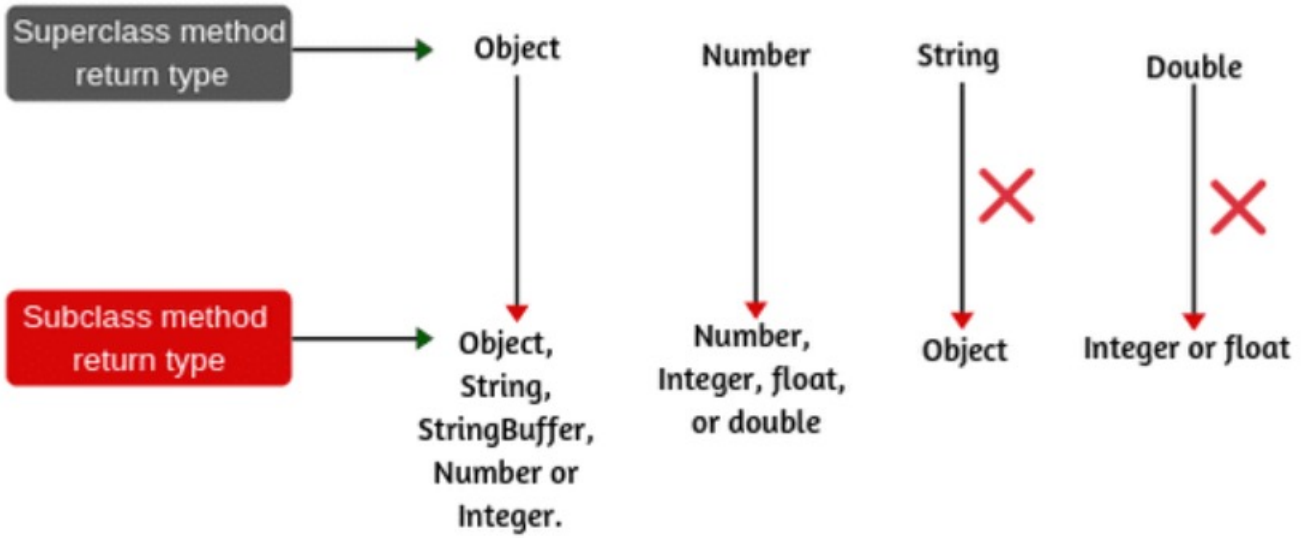
Access Level of Overriding Method

- The access level of an overriding method can be higher, but not lower than that of the overridden method (package-private < protected < public)
- The access modifier for an overriding method can allow more, but not less, access than the overridden method (e.g., a protected instance method in the superclass can be made public, but not private, in the subclass.)

The subclass must present *at least the same* interface as the superclass. Making protected/public things less visible would violate this idea

Return Type of Overriding Method

- When overriding a method in a subclass, the signature must match, the return type should:
  - Either be the same
  - Or be a subtype of the return type of the superclass's method (an overridden method can have a more specific return type)



Method Overriding

- Overriding requires an inheritance relation.
- Overriding is for accessible instance methods
- Static methods CANNOT be overridden, but can be hidden (see next lecture)
- Class fields CANNOT be overridden, but can be hidden (later)

Using the `super` Keyword

- (Usage #1) The `super` keyword can be used to invoke a superclass's constructor (as illustrated by our earlier example)
- (Usage #2) If your method overrides its superclass's method, you can invoke that method of the superclass using the keyword `super`.
- (Usage #3) `super` can be used to refer to instance variables of the parent class.

The super Keyword: usage #3

```
class Animal{
    String color="white";
}

class Dog extends Animal{

    String color="black";

    void printColor(){
        //print color of Dog: black
        System.out.println(color);
        //print color of Animal: white
        System.out.println(super.color);
    }

}
```

What if we remove this line?

```
Dog dog = new Dog();

dog.printColor();
```

- Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.
- Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through `super`

Lookup Order

```
class Grandpa {

    String name = "Name: grandpa";
    int age = 99;
}

class Dad extends Grandpa {

    String name = "Name: dad";
    int height = 180;
}

class Me extends Dad {

    String name = "Name: me";
}
```

```
public static void main(String[] args) {

    Me m = new Me();

    System.out.println(m.name);
    System.out.println(m.height);

    System.out.println(m.age);
}
```

```
Name: me
180
99
```

Inheritance in a Nutshell

- **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, including static/instance fields and methods, but not constructors, though the private superclass members are hidden (i.e., cannot be accessed) from the subclass.

Inheritance in a Nutshell

- 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.

- By doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them all by yourself from scratch.