# Object-Oriented Programming: Polymorphism

Java™ How to Program, 11th Edition

Instructor: Zhuozhao Li

# Objectives

- Polymorphism
- Override
- Abstract and concrete classes
- Determine an object's type
- Interface

# Polymorphism (多态)

- The word **polymorphism** is used in various disciplines (学科) to describe situations where something occurs in several different forms

Light-morph jaguar （美洲虎）

Dark-morph jaguar

***Biology example:*** About 6% of the South American population of jaguars are dark-morph jaguars.

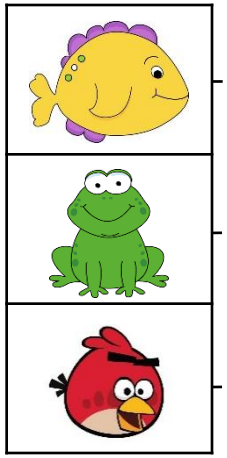
# Polymorphism in OOP

- In Java, **polymorphism** is the ability of an object to take on many forms
- Objects of different types can be accessed through the same interface. Each type can provide its own, independent implementation of this interface.
- **Example:** Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent three types of animals under study.
- Each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x*-*y* coordinates. Each subclass implements (overrides) method `move`.

# Polymorphism in OOP

```
Animal[] animals = prepareAnimals();
for(Animal a : Animals) {
    a.move();
}
```

Each specific type of `Animal` responds to a `move` message in a unique way



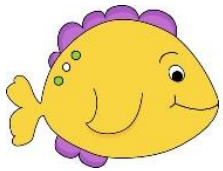
A fish might swim 3 feet

A frog might jump 5 feet

A bird might fly 10 feet

# Polymorphism in OOP

- An object of subclass can be treated as an object of the super class.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.



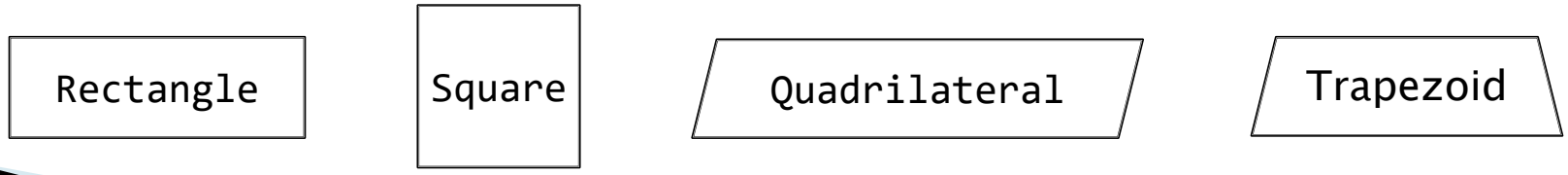
. . .

# Polymorphism in OOP

- Polymorphism enables you to write programs to process objects that share the same superclass as if they're all objects of the superclass.
- With polymorphism, we can design and implement *extensible* systems
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes (e.g., the part that creates the corresponding objects).

# Another Example: Quadrilaterals

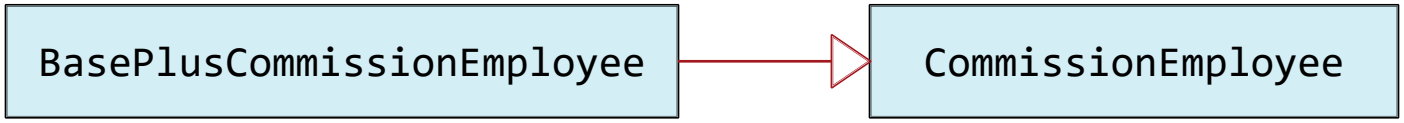
- If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
- Any operation (e.g., calculating area) that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`.
- These operations can also be performed on other `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`.
- Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—at execution time, the correct subclass version of the method is called, based on the exact type of the object.



# Polymorphic Behavior

- **All Java objects are polymorphic** since any object will pass the **IS-A test** for at least their own type and the class `Object`
  - A bird is an instance of `Bird` class, also an instance of `Animal` and `Object`

- Earlier, when we write programs, we aim super class variables at superclass objects and subclass variables at subclass objects




```
CommissionEmployee employee1 = new CommissionEmployee(…);
BasePlusCommissionEmployee employee2 = new BasePlusCommissionEmployee(…);
```

Such assignments are natural

# Polymorphic Behavior

- In Java, we can also aim a superclass reference at a subclass object (the most common use of polymorphism)

```
CommissionEmployee employee = new BasePlusCommissionEmployee(…);
```

This is totally fine due to the is-a relationship (an instance of the subclass is also an instance of superclass)

```
BasePlusCommissionEmployee employee = new CommissionEmployee(…);
```

This will not compile, the is-a relationship only applies up the class hierarchy

# Polymorphic Behavior

- Then the question comes…

```
CommissionEmployee employee = new BasePlusCommissionEmployee(…);
double earnings = employee.earnings();
```

***Question:*** *Which version of* `earnings()` *will be invoked? The one in the superclass or the one overridden by the subclass?*

- Which method is called is determined by **the type of the referenced object**, not the type of the variable.
- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.

# Demo

Variable refers to a CommissionEmployee object, so that class's `toString()` method will be called

```
public class PolymorphismTest {

    public static void main(String[] args) {

        CommissionEmployee commissionEmployee = new CommissionEmployee(
                "Sue", "Jones", "222-22-2222", 10000, .06);

        BasePlusCommissionEmployee basePlusCommissionEmployee = new BasePlusCommissionEmployee(
                "Bob", "Lewis", "333-33-3333", 5000, .04, 300);

        System.out.printf("%s %s:\n\n%s\n\n",
                "Call CommissionEmployee's toString with superclass reference",
                "to superclass object", commissionEmployee.toString());

        System.out.printf("%s %s:\n\n%s\n\n",
                "Call BasePlusCommissionEmployee's toString with subclass",
                "reference to subclass object",
                basePlusCommissionEmployee.toString());
```

Similarly, `BasePlusCommissionEmployee`'s `toString()` method will be called

# Demo

```
Call CommissionEmployee's toString with superclass reference to
superclass object:

commission employee: Sue Johes
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

# Demo

```
CommissionEmployee commissionEmployee2 = basePlusCommissionEmployee;
System.out.printf("%s %s:\n\n%s\n",
        "Call BasePlusCommissionEmployee's toString with superclass",
        "reference to subclass object", commissionEmployee2.toString());
```

Although the variable's type is `CommissionEmployee`, it refers to an object of `BasePlusCommissionEmployee`, so the subclass's `toString()` method will be called

```
Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

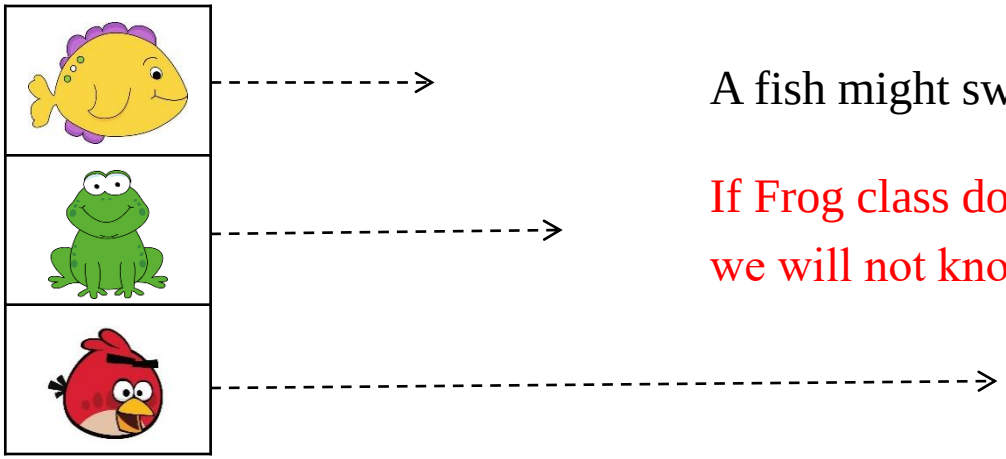
# Polymorphic Behavior (Cont.)

- When the Java compiler encounters a method call made through a variable, it determines if the method can be called by checking the variable's class type.
  - If that class contains the proper method declaration (or inherits one), the call will be successfully compiled

- At execution time, the type of the object to which the variable refers determines the actual method to use (JVM will take care of this).
  - This process is called **dynamic binding**. Binding means "associating method calls to the appropriate method body".

# Polymorphic Behavior (Cont.)

- What if the subclasses do not override the superclass's method to implement its own specific version (i.e., use the inherited one as is)?



A fish might swim 3 feet

If Frog class does not override move method, we will not know how frogs would move…

A bird might fly 10 feet

Can we force a subclass to override a method inherited from superclass?

Yes, we can leverage the power of abstract class.

# Concrete Classes

- All classes we have defined so far provide implementations of every method they declare (some of the implementations can be inherited)
- They are called "**concrete classes**"
- Concrete classes can be used to instantiate objects

# Abstract Classes

- Sometimes it's useful to declare "incomplete" classes for which you never intend to create objects.
- Used only as superclasses in inheritance hierarchies
- They are called "**abstract classes**", cannot be used to instantiate objects
- Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract

# Abstract Classes

- An abstract class provides a superclass from which other classes can inherit and thus share a common design. Not all hierarchies contain abstract classes.
- Programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of subclass types (i.e., program in general not in specific)
  - `moveAnAnimal(Animal a) { … }` (suppose Animal is an abstract class)
  - When called, such a method can receive an object of any concrete class that directly or indirectly extends the abstract superclass `Animal`.

# Declaring Abstract Classes

- You make a class abstract by declaring it with keyword `abstract`.
- An abstract class normally contains one or more abstract methods, which are declared with the keyword `abstract` and provides no implementations.

```
public abstract class Animal {
    public abstract void move();
    // ...
}
```

→ Be careful, no brackets {}

# Abstract Method

```
public abstract class Animal {
    public abstract void move();
}
```

- Abstract methods have the same visibility rules as normal methods, except that they cannot be private.
- Private abstract methods make no sense since abstract methods are intended to be overridden by subclasses.
- Abstract methods have no implementations because the abstract classes are too general and only specify the common interfaces of subclasses
  - **Think about this:** How can an `Animal` class provide an appropriate implementation for `move()` method without knowing the specific type of the animal? Every type of animal moves in a different way.

# Abstract Classes (cont.)

- A class that contains abstract methods must be declared as `abstract` even if that class contains some concrete methods.
- If a subclass does not implement all abstract methods it inherits from the superclass, the subclass must also be declared as `abstract` and thus cannot be used to instantiate objects.
  - Using abstract classes addresses our earlier problem "the `Frog` class does not override `Animal`'s `move()` method to define specific behaviors of frogs".
- Constructors and `static` methods cannot be declared `abstract` (constructors are not inherited, non-`private static` methods are inherited but cannot be overridden)

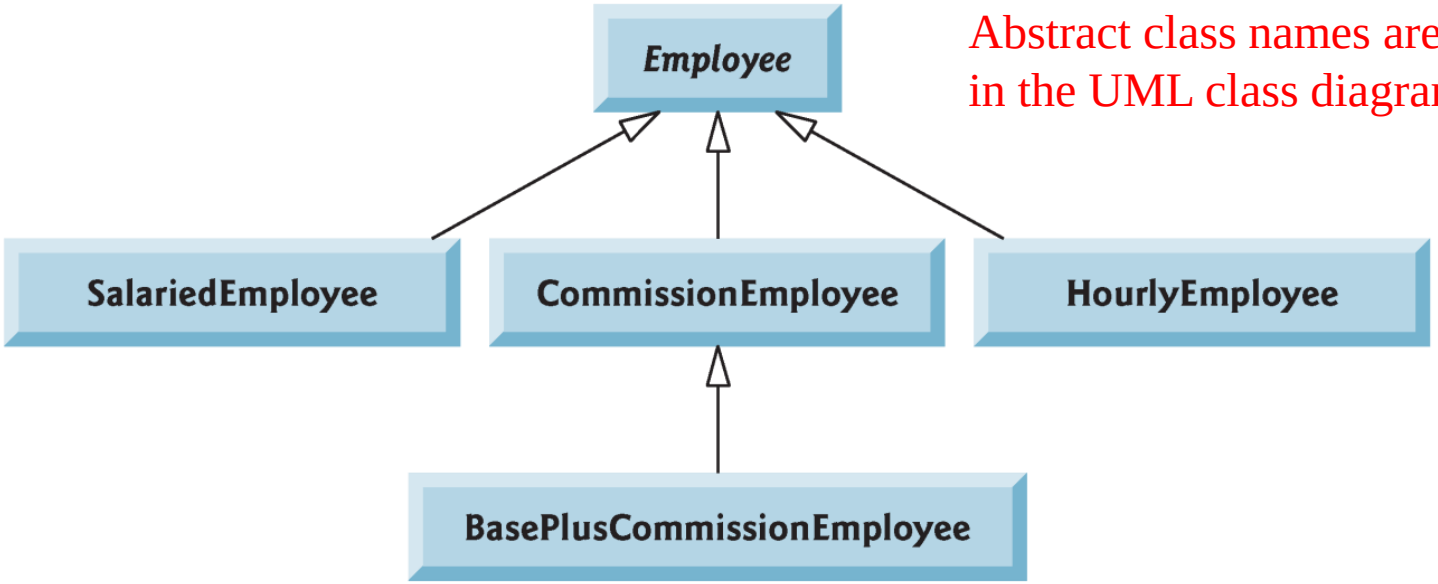
# Abstract Classes (cont.)

- Although abstract classes cannot be used to instantiate objects, they can be used to declare variables
- Abstract superclass variables can hold references to objects of any concrete class derived from them.
  - `Animal animal = new Frog(…); // assuming Animal is abstract`
- Such practice is commonly adopted to manipulate objects polymorphically.
- Note that we can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

# Case Study: A Payroll System Using Polymorphism

- The company pays its four types of employees on a weekly basis.
  - **Salaried employees** get a fixed weekly salary regardless of working hours
  - **Hourly employees** are paid for each hour of work and receive overtime pay (i.e., 1.5x their hourly salary rate) for after 40 hours worked
  - **Commission employees** are paid a percentage of their sales
  - **Salaried-commission employees** get a base salary + a percentage of their sales.
- For the current pay period, the company has decided to reward **salaried-commission employees** by adding 10% to their base salaries.
- The company wants to write a Java application that performs its payroll calculations polymorphically.

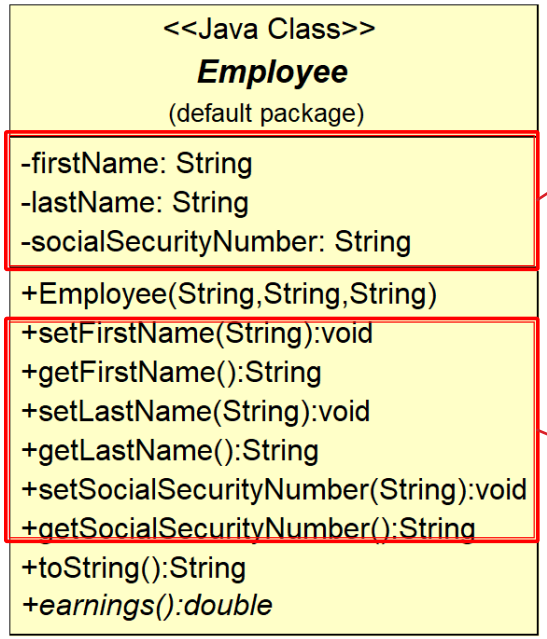
# The Design: Main Classes




Abstract class names are italicized in the UML class diagrams

# The Employee Abstract Class

- Abstract superclass `Employee` declares the "interface": the set of methods that a program can invoke on all `Employee` objects.

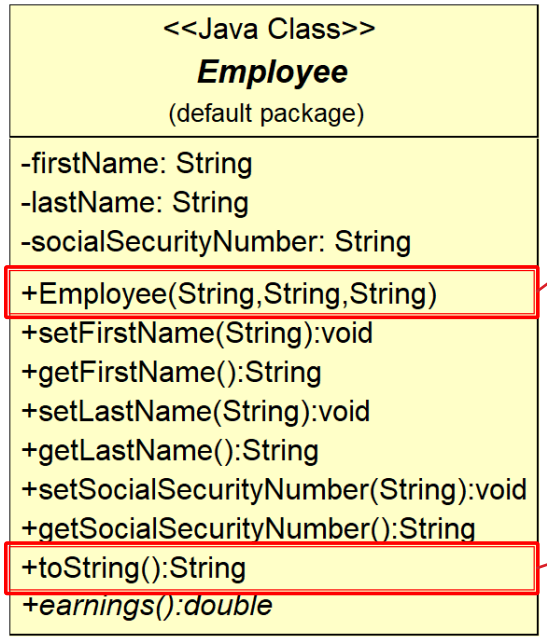



Each employee has a first name, a last name and a social security number. This applies to all employee types.

Set and get methods for each field. These methods are concrete and the same for all employee types.

# The Employee Abstract Class

- Abstract superclass `Employee` declares the "interface": the set of methods that a program can invoke on all `Employee` objects.

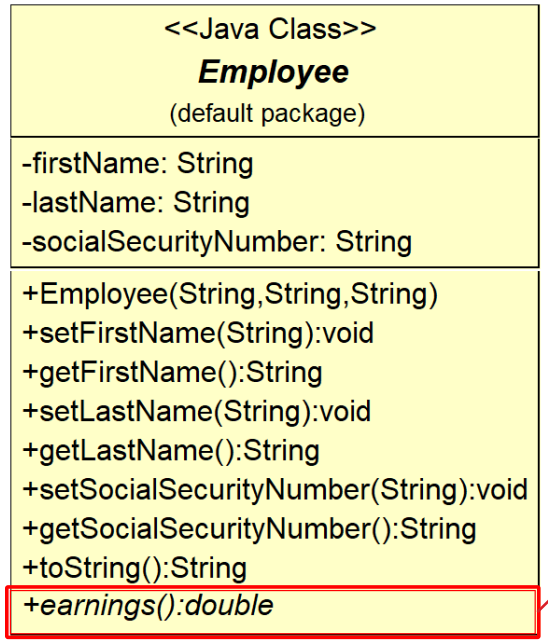



A constructor for initializing the three fields

Represent the employee's basic information as a string

# The `Employee` Abstract Class

- Abstract superclass `Employee` declares the "interface": the set of methods that a program can invoke on all `Employee` objects.

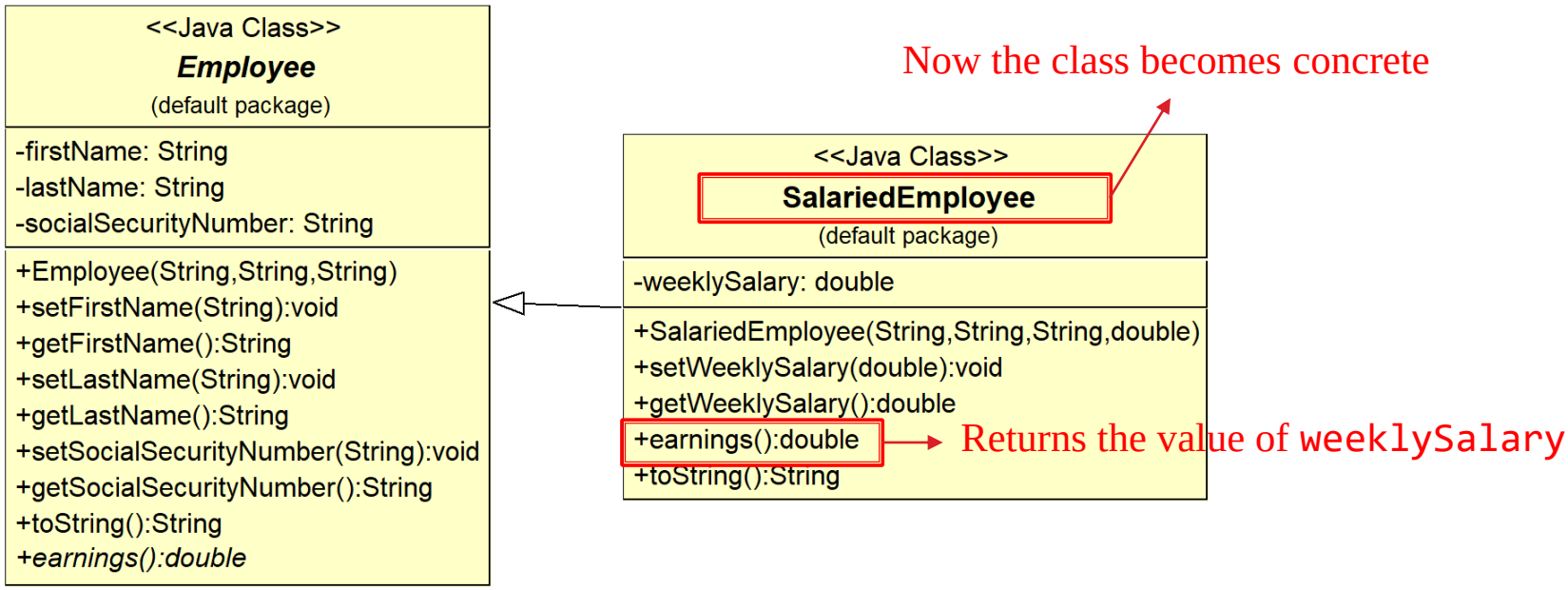



Abstract method names are italicized

Abstract method that needs to be implemented by the subclasses (the `Employee` class does not have enough information to do the calculation)

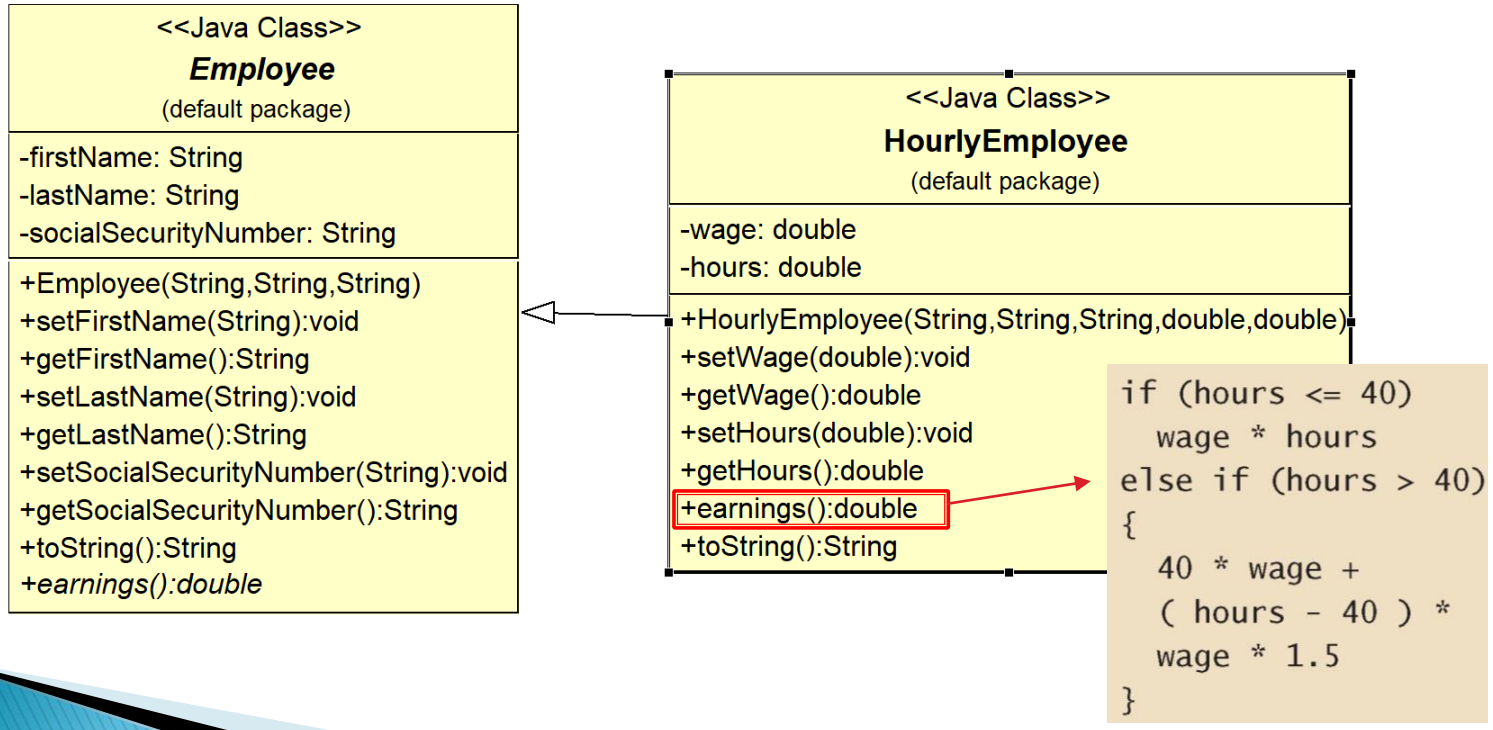
# The SalariedEmployee Class

- Defines a new field `weeklySalary`, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.



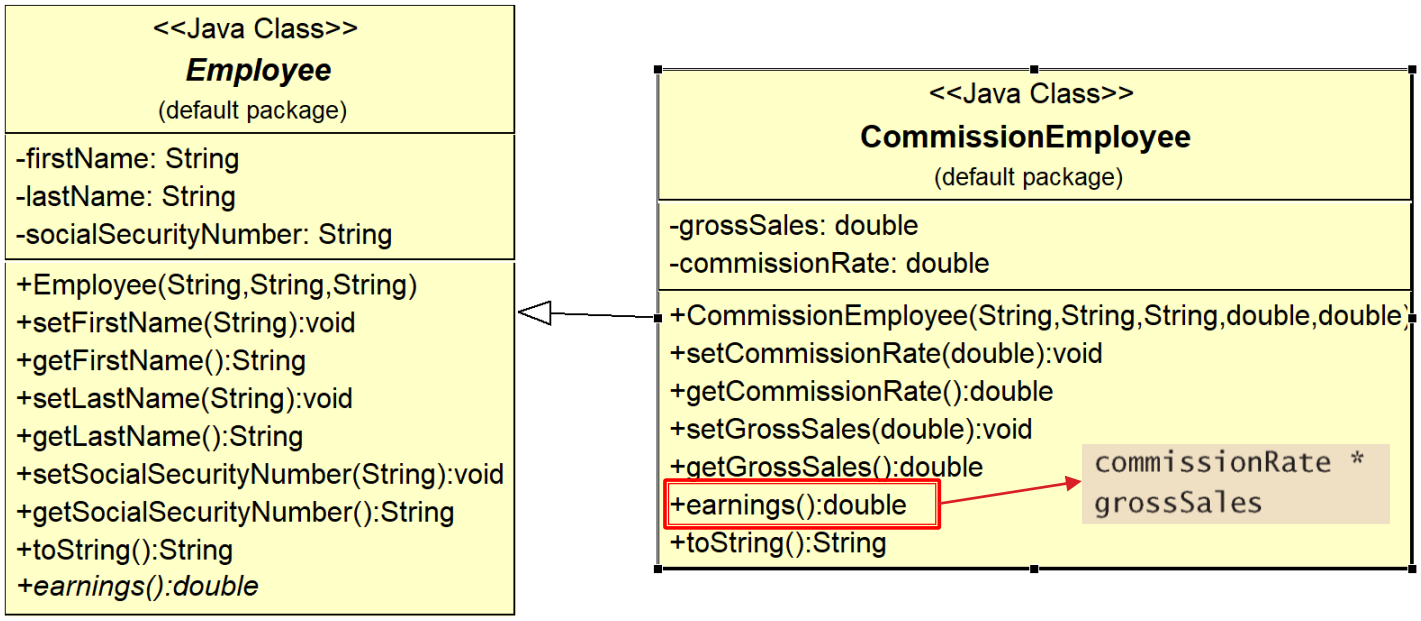
# The HourlyEmployee Class

- Defines two new fields, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.



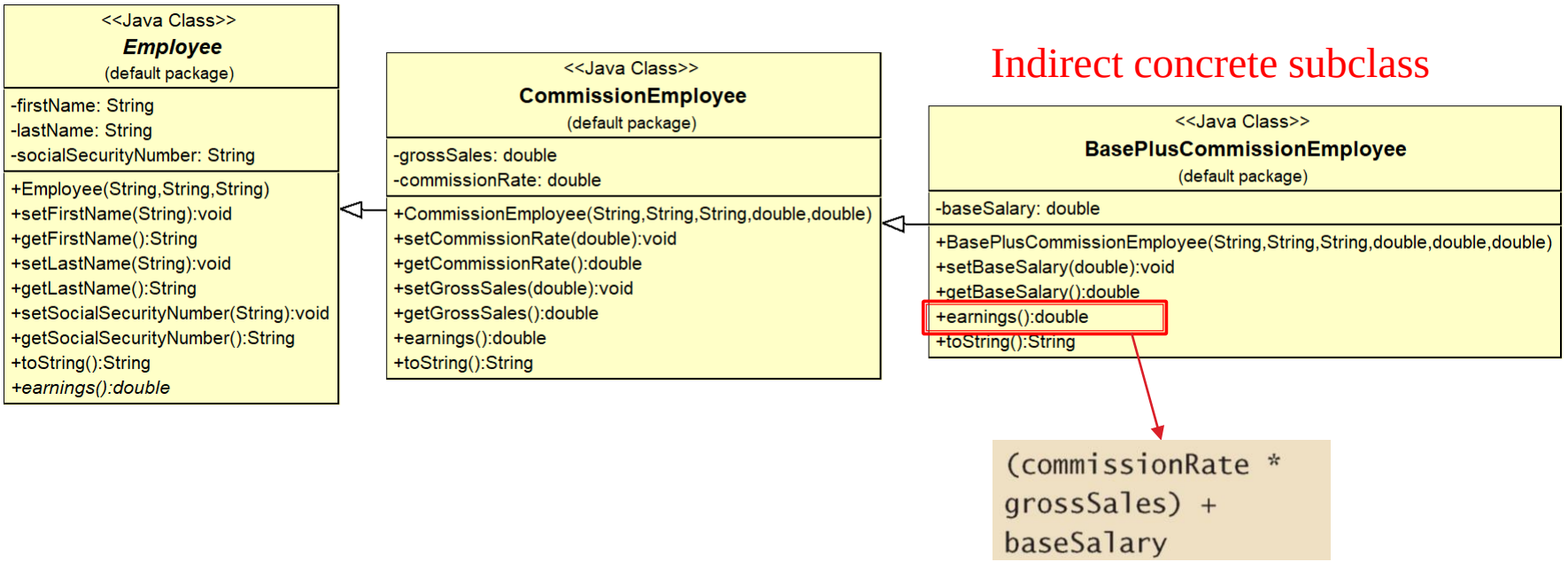
# The CommissionEmployee Class

- Defines two new fields, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.



# The BasePlusCommissionEmployee Class

- Extends `CommissionEmployee`. Defines a new field, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.



# The Employee Abstract Class

```
// Employee abstract superclass.
public abstract class Employee {
    private final String firstName;
    private final String lastName;
    private final String socialSecurityNumber;

    // constructor
    public Employee(String firstName, String lastName, String
    socialSecurityNumber) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.socialSecurityNumber = socialSecurityNumber;
    }
```

Although abstract classes cannot be used to instantiate objects, they can have constructors.

# The Employee Abstract Class

```
public String getFirstName() { return firstName; }
public String getLastName() { return lastName; }

public String getSocialSecurityNumber() {
   return socialSecurityNumber;
}

// return String representation of Employee object
@Override
public String toString() {
   return String.format("%s %s%nsocial security number: %s",
      getFirstName(), getLastName(), getSocialSecurityNumber());
}

// abstract method must be overridden by concrete subclasses
public abstract double earnings(); // no implementation here

} // end abstract class Employee
```

# The SalariedEmployee Class

```
// SalariedEmployee concrete class extends abstract class Employee.
public class SalariedEmployee extends Employee {
   private double weeklySalary;

   // constructor
   public SalariedEmployee(String firstName, String lastName, String
socialSecurityNumber, double weeklySalary) {
      super(firstName, lastName, socialSecurityNumber);
      setWeeklySalary(weeklySalary);
   }

   public void setWeeklySalary(double weeklySalary) {
      this.weeklySalary = weeklySalary < 0.0 ? 0.0 : weeklySalary;
   }
```

Initialize private fields that are not inherited

# The SalariedEmployee Class

```
    // return salary
    public double getWeeklySalary() { return weeklySalary; }

    // calculate earnings; override abstract method earnings in Employee
    @Override
    public double earnings() {
       return getWeeklySalary();
    }

    // return String representation of SalariedEmployee object
    @Override
    public String toString() {
       return String.format("salaried employee: %s%n%s: $%,.2f",
          super.toString(), "weekly salary", getWeeklySalary());
    }
} // end class SalariedEmployee
```

**Code reuse, good practice**

# The HourlyEmployee Class

```
// HourlyEmployee class extends Employee
public class HourlyEmployee extends Employee {
    private double wage;
    private double hours;

    // constructor
    public HourlyEmployee(String firstName, String lastName, String
socialSecurityNumber, double wage, double hours) {
        super(firstName, lastName, socialSecurityNumber);
        setWage(wage);
        setHours(hours);
    }
```

# The HourlyEmployee Class

```
public void setWage(double wage) {
    this.wage = wage < 0.0 ? 0.0 : wage;
}

public void setHours(double hours) {
    this.hours = ((hours < 0.0) && (hours > 168)) ? 0.0 : hours;
}

// return salary
public double getWage() { return wage; }

// return salary
public double getHours() { return hours; }
```

# The HourlyEmployee Class

```
// calculate earnings; override abstract method earnings in Employee
   @Override
   public double earnings() {
      if (getHours() <= 40) {// no overtime
         return getWage() * getHours();
      } else {
         return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
      }
   }

   // return String representation of HourlyEmployee object
   @Override
   public String toString() {
      return String.format("hourly employee: %s%n%s: $%,.2f; %s:
%,.2f",
         super.toString(), "hourly wage", getWage(),
         "hours worked", getHours());
   }
} // end class SalariedEmployee
```

**Code reuse, good practice**

# The CommissionEmployee Class

```
public class CommissionEmployee extends Employee {
   private double grossSales;
   private double commissionRate;

   public CommissionEmployee(String firstName, String lastName, String
socialSecurityNumber,
                      double sales, double rate) {
      super(firstName, lastName, socialSecurityNumber);
      setGrossSales(sales); // data validation
      setCommissionRate(rate); // data validation
   }

   public void setGrossSales(double sales) {
      grossSales = (sales < 0.0) ? 0.0 : sales;
   }

   public double getGrossSales() {  return grossSales;  }
```

```
public void setCommissionRate(double rate) {
    commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
}

public double getCommissionRate() {  return commissionRate;  }

@Override
public double earnings() {
    return getCommissionRate() * getGrossSales();
}

// return String representation of CommissionEmployee object
@Override
public String toString() {
    return String.format("%s: %s%n%s: $%,.2f; %s: %.2f",
        "commission employee", super.toString(),
        "gross sales", getGrossSales(),
        "commission rate", getCommissionRate());
}
}
```

# The BasePlusCommissionEmployee Class

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

   public BasePlusCommissionEmployee(String firstName, String
lastName, String socialSecurityNumber,
                      double sales, double rate, double salary) {
      super(firstName, lastName, socialSecurityNumber, sales, rate);
      setBaseSalary(salary);
   }

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

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

```
public double earnings() {
        return getBaseSalary() + super.earnings();
    }

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

# Putting Things Together

```
// Employee hierarchy test program.
public class PayrollSystemTest {
   public static void main(String[] args) {
      // create subclass objects
      SalariedEmployee salariedEmployee = new
            SalariedEmployee("John", "Smith", "111-11-1111", 800.00);

      HourlyEmployee hourlyEmployee = new
            HourlyEmployee("Karen", "Price", "222-22-2222", 16.75, 40);

      CommissionEmployee commissionEmployee = new
            CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06);

      BasePlusCommissionEmployee basePlusCommissionEmployee = new
            BasePlusCommissionEmployee("Bob", "Lewis", "444-44-4444",
5000, .04, 300);
```

Create an object of each of the four concrete classes

Manipulates these objects non-polymorphically, via variables of each object's own type

```
System.out.println("Employees processed individually:");

      System.out.printf("%n%s%n%s: $%,.2f%n%n", salariedEmployee, "earned",
salariedEmployee.earnings());
      System.out.printf("%s%n%s: $%,.2f%n%n", hourlyEmployee, "earned",
hourlyEmployee.earnings());
      System.out.printf("%s%n%s: $%,.2f%n%n", commissionEmployee, "earned",
commissionEmployee.earnings());
      System.out.printf("%s%n%s: $%,.2f%n%n", basePlusCommissionEmployee,
"earned" basePlusCommissionEmployee.earnings());
```

Manipulates these objects polymorphically, using an array of `Employee` variables

```
// create four-element Employee array
    Employee[] employees = new Employee[4];
    // initialize array with Employees
    employees[0] = salariedEmployee;
    employees[1] = hourlyEmployee;
    employees[2] = commissionEmployee;
    employees[3] = basePlusCommissionEmployee;
    System.out.printf("Employees processed polymorphically:%n%n");

    // generically process each element in array employees
    for (Employee currentEmployee : employees) {
       System.out.println(currentEmployee); // invokes toString
```

All calls to `toString` are resolved at execution time, based on the type of the object to which `currentEmployee` refers (dynamic binding or late binding)

The operator `instanceof` determines the object's type at execution time (IS-A test)

```
// determine whether element is a BasePlusCommissionEmployee
if (currentEmployee instanceof BasePlusCommissionEmployee)
{
   // downcast Employee reference to
   // BasePlusCommissionEmployee reference
   BasePlusCommissionEmployee employee =
      (BasePlusCommissionEmployee) currentEmployee;
   employee.setBaseSalary(1.10 * employee.getBaseSalary());

   System.out.printf("new base salary with 10%% increase is:
      $%,.2f%n", employee.getBaseSalary());
} // end if

System.out.printf("earned $%,.2f%n%n",
   currentEmployee.earnings());
}
```

Downcasting

Method call resolved at execution time

Without downcasting, the get and set methods cannot be invoked.

Superclass reference can be used to invoke only methods of the superclass

Finally, the program polymorphically determines and outputs the type of each object in the `Employee` array

```
// get type name of each object in employees array
      for (int j = 0; j < employees.length; j++)
         System.out.printf("Employee %d is a %s%n", j,
employees[j].getClass().getName());
   } // end main
} // end class PayrollSystemTest
```

Every Java object knows its own class and can access this information through the `getClass` method, which all classes inherit from class `Object`

The `getClass` method returns an object of type `java.lang.Class`, which contains information about the object's type, including its class name (can be retrieved via calling `getName` method)

Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00

Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

```
Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

# Assignments Between Superclass and Subclass Variables (Summary)

- Assigning a superclass object's reference to a superclass variable is **natural**.
- Assigning a subclass object's reference to a subclass variable is **natural**.
- Assigning a subclass object's reference to a superclass variable is **safe**, because the subclass object *is also an object of its superclass* **(Java objects are polymorphic)**.
  - The superclass variable can be used to refer only to superclass members.
  - If a program refers to subclass-only members through the superclass variable, the compiler reports errors.

# Assignments Between Superclass and Subclass Variables (Summary)

- Attempting to assign a superclass object's reference to a subclass variable is a **compilation error**.
- To avoid this error, the superclass object's reference must be cast to a subclass type explicitly.
- At execution time, if the object to which the reference refers is not a subclass object, an exception will occur.
- Use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.