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.

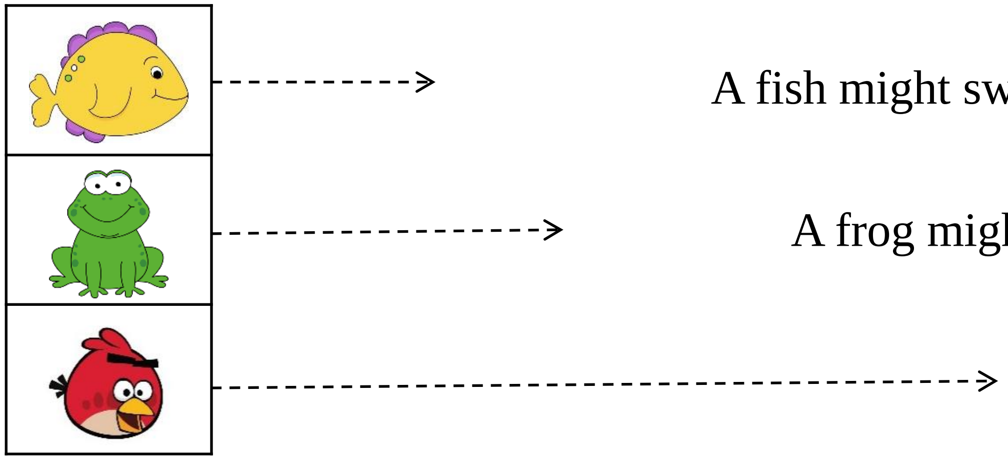
Polymorphism in OOP

- 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



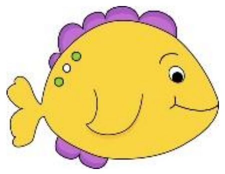
A fish might swim 3 feet

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.



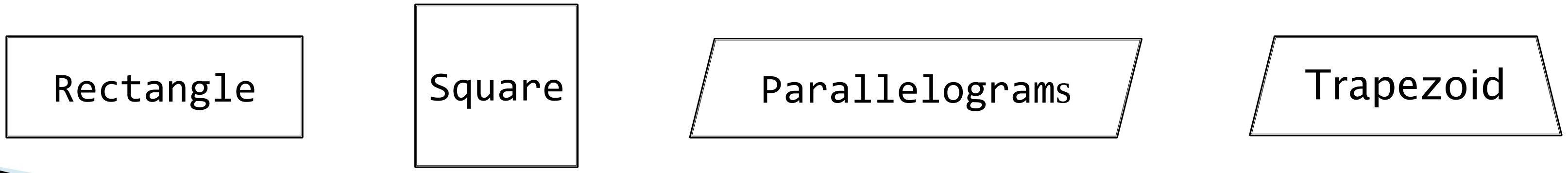
. . .

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).

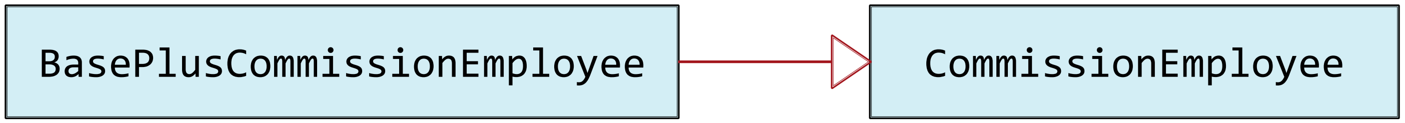
Another Example: Quadrilaterals

- 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…

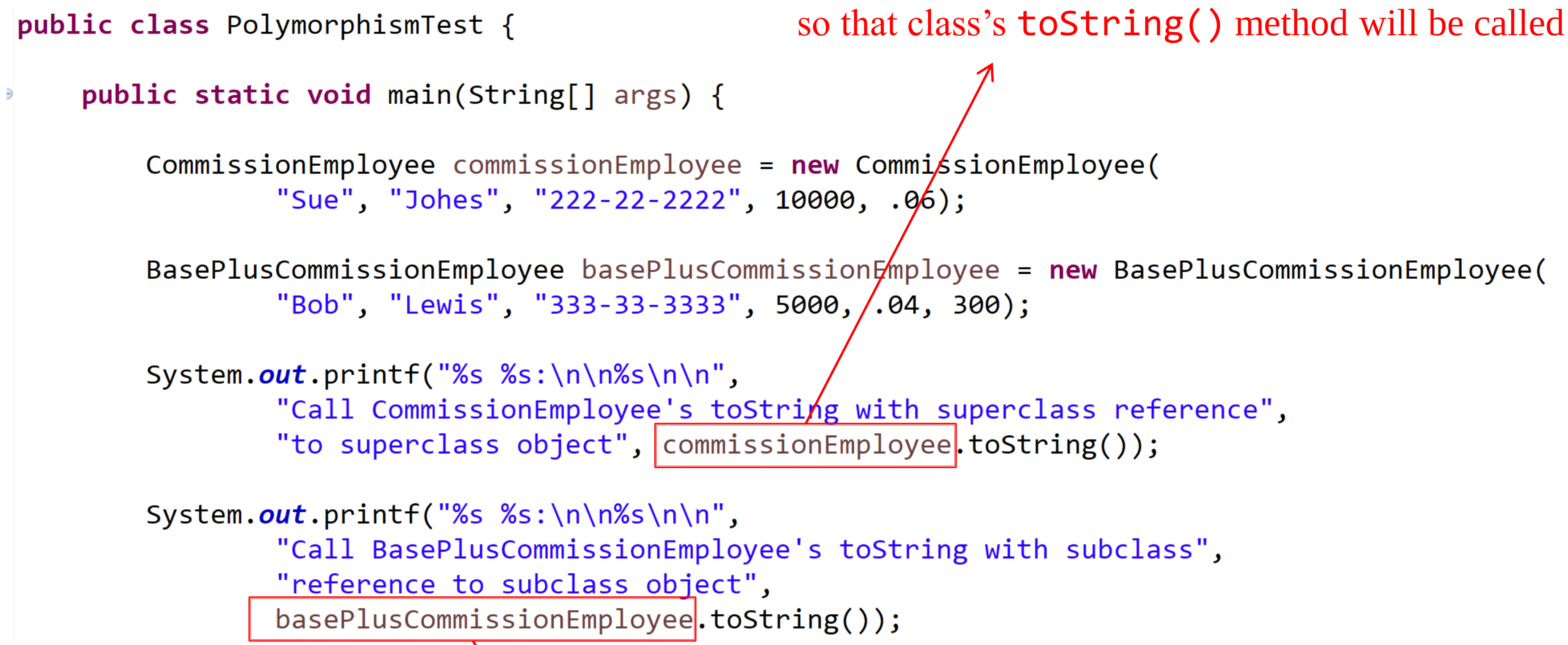
```
CommissionEmployee employee = new BasePlusCommissionEmployee(…);
double earnings = employee.earnings();
```

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", "Johes", "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

```java
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
```

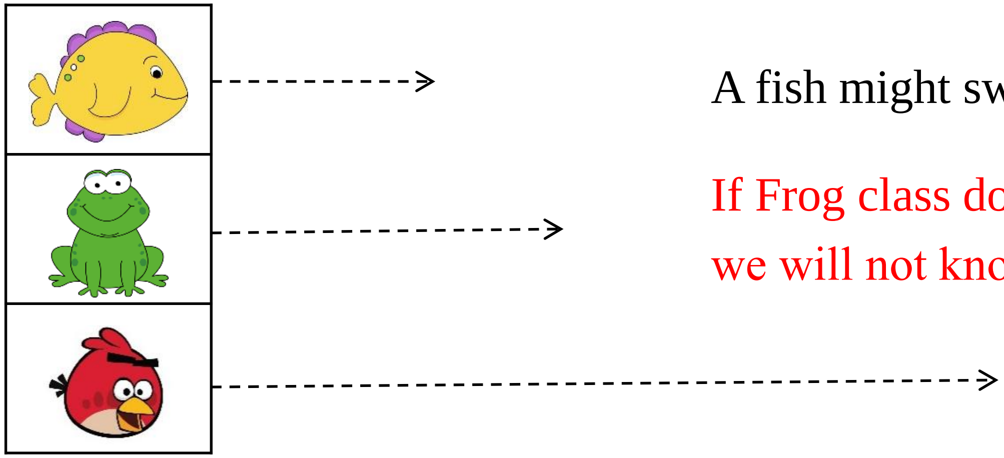
Polymorphic Behavior (Cont.)

- 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();
    // ...
}
```

$\longrightarrow$ 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)

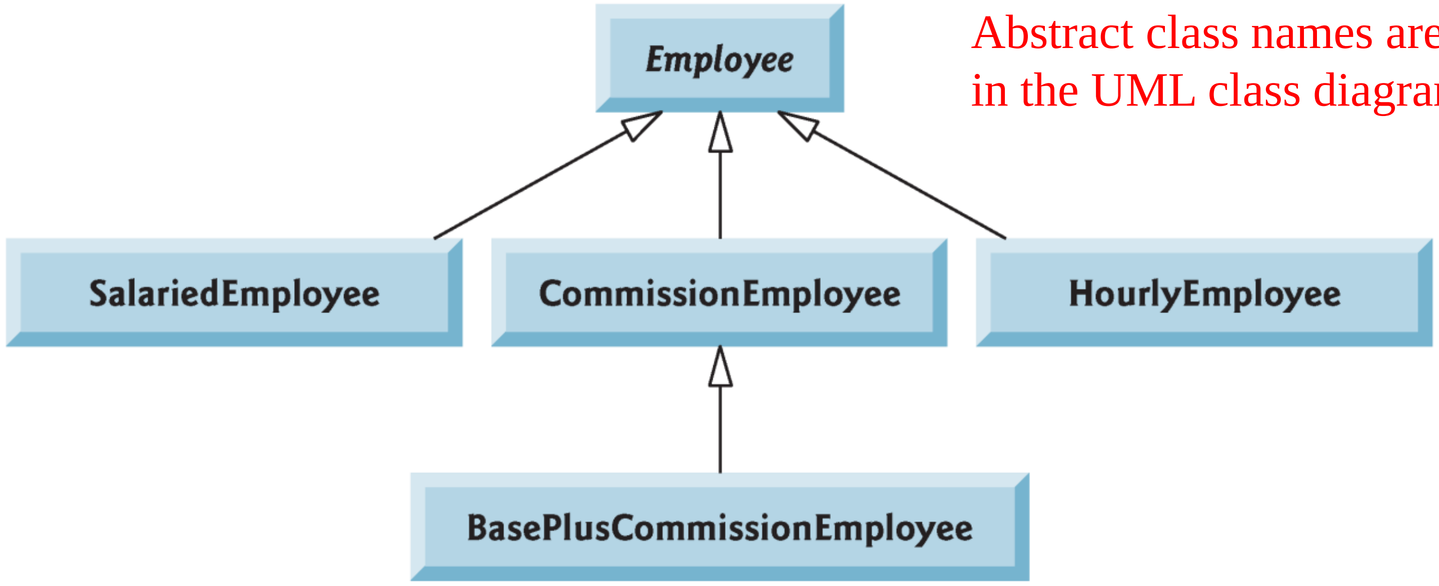
Abstract Classes (cont.)

- 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.

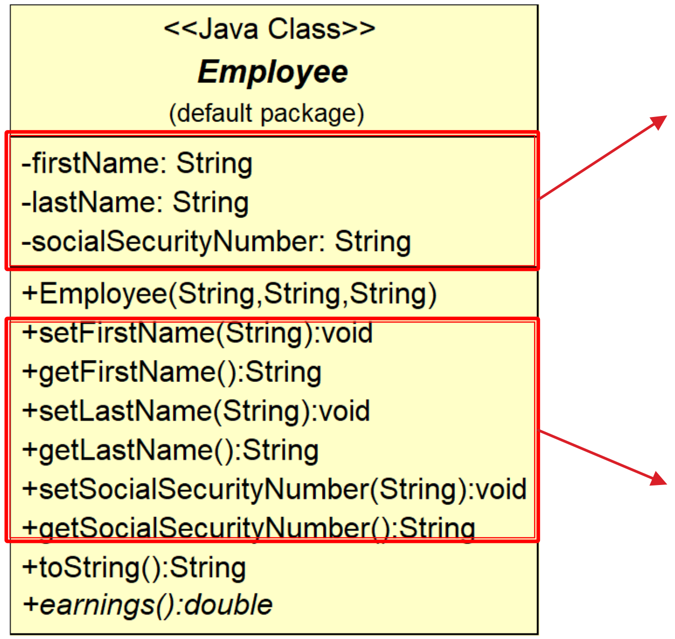
The Design: Main Classes



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.



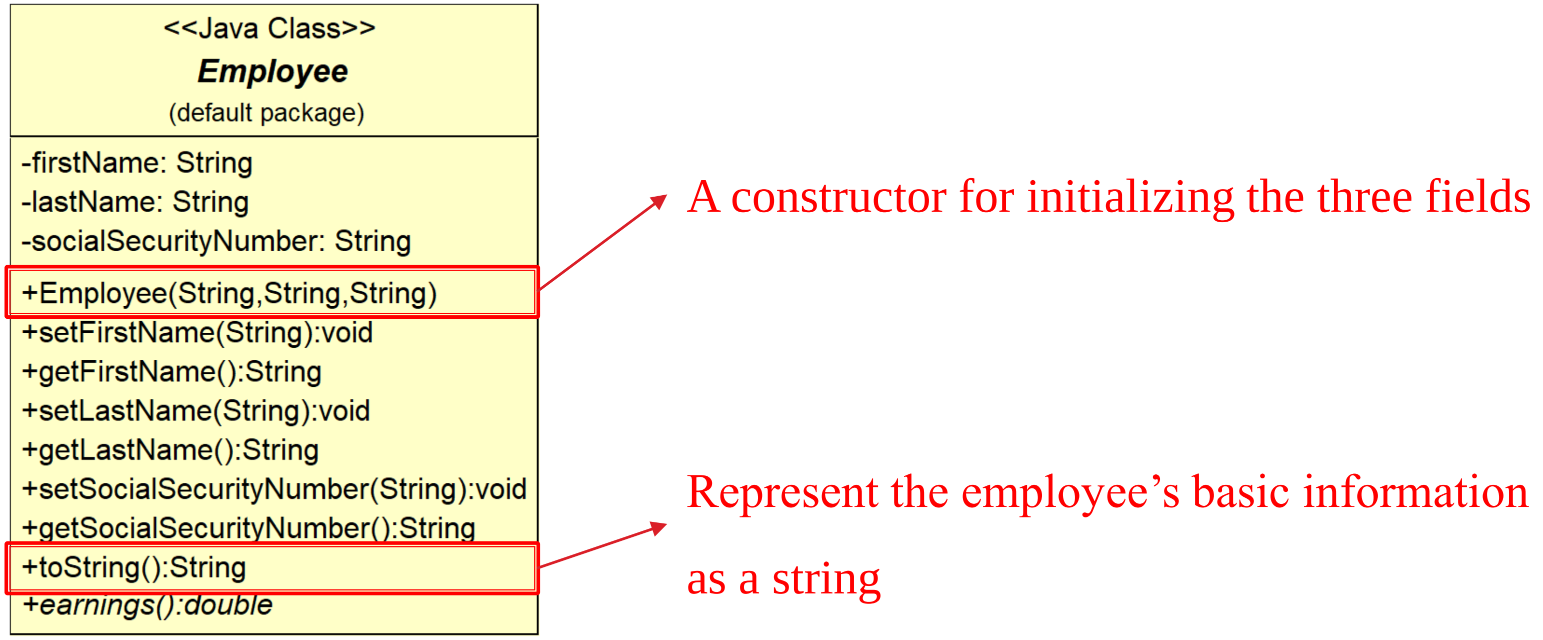
<<Java Class>>
Employee
(default package)
-firstName: String
-lastName: String
-socialSecurityNumber: String
+Employee(String,String,String)
+setFirstName(String):void
+getFirstName():String
+setLastName(String):void
+getLastName():String
+setSocialSecurityNumber(String):void
+getSocialSecurityNumber():String
+toString():String
+earnings():double

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

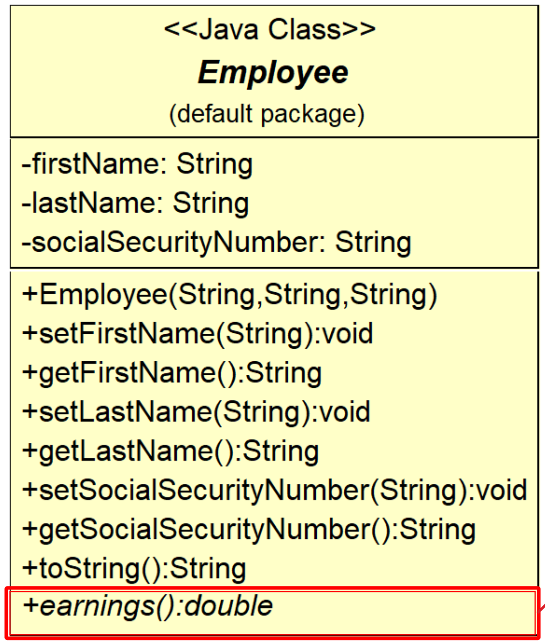
- Abstract superclass `Employee` declares the “interface”: the set of methods that a program can invoke on all `Employee` objects.



<<Java Class>>
Employee
(default package)
-firstName: String
-lastName: String
-socialSecurityNumber: String
+Employee(String,String,String)
+setFirstName(String):void
+getFirstName():String
+setLastName(String):void
+getLastName():String
+setSocialSecurityNumber(String):void
+getSocialSecurityNumber():String
+toString():String
+earnings():double
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.

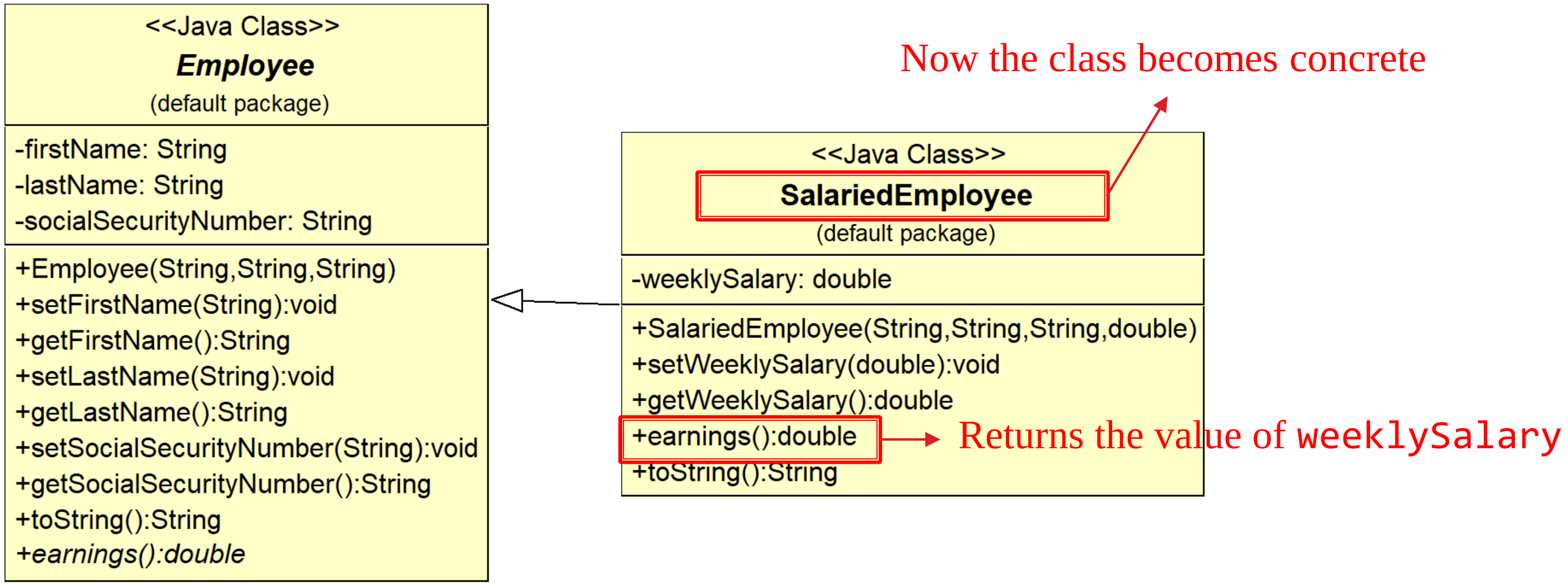

<<Java Class>>
Employee
(default package)
-firstName: String
-lastName: String
-socialSecurityNumber: String
+Employee(String,String,String)
+setFirstName(String):void
+getFirstName():String
+setLastName(String):void
+getLastName():String
+setSocialSecurityNumber(String):void
+getSocialSecurityNumber():String
+toString():String
+earnings():double

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

Abstract method names are italicized

The SalariedEmployee Class

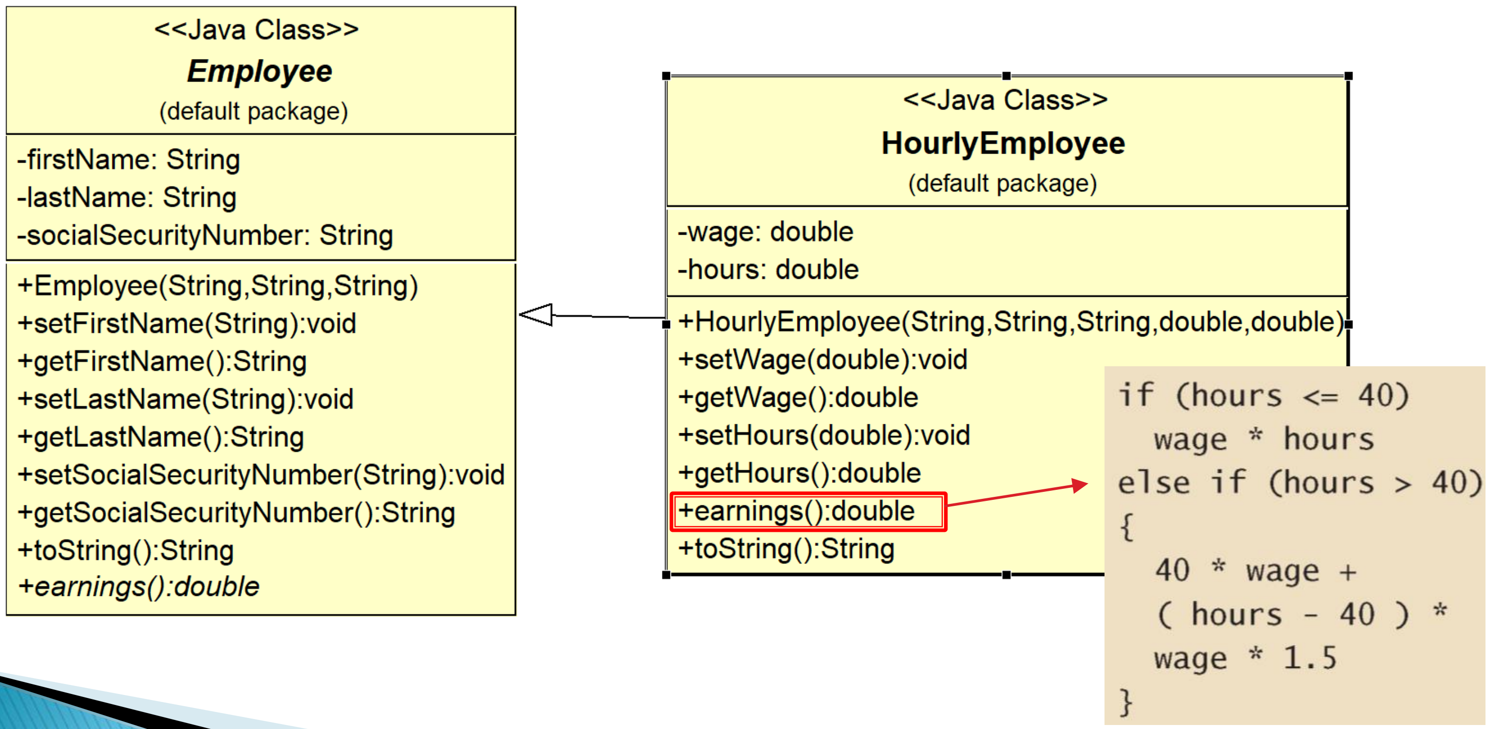
- Defines a new field `weeklySalary`, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.



<<Java Class>>
Employee
(default package)
-firstName: String
-lastName: String
-socialSecurityNumber: String
+Employee(String,String,String)
+setFirstName(String):void
+getFirstName():String
+setLastName(String):void
+getLastName():String
+setSocialSecurityNumber(String):void
+getSocialSecurityNumber():String
+toString():String
+earnings():double
<<Java Class>>
SalariedEmployee
(default package)
-weeklySalary: double
+SalariedEmployee(String,String,String,double)
+setWeeklySalary(double):void
+getWeeklySalary():double
+earnings():double
+toString():String
Now the class becomes concrete
Returns the value of weeklySalary

The HourlyEmployee Class

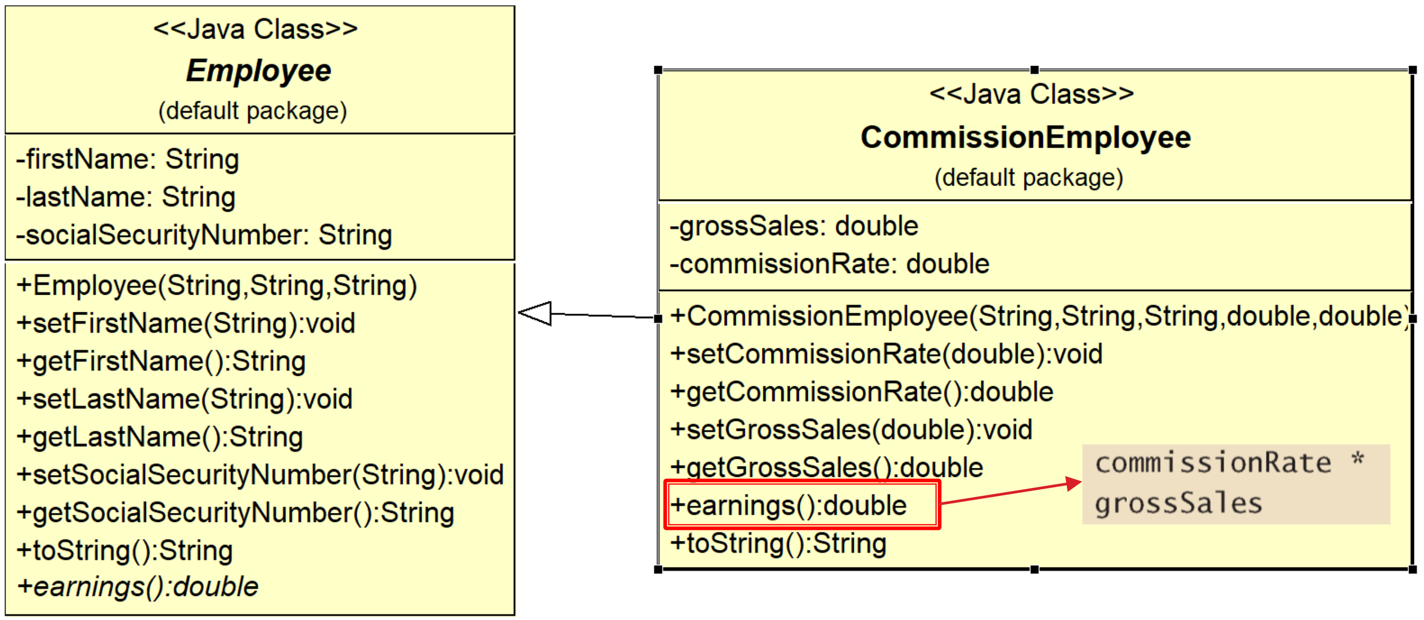
- Defines two new fields, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.



```
if (hours <= 40)
  wage * hours
else if (hours > 40)
{
  40 * wage +
  ( hours - 40 ) *
  wage * 1.5
}
```

The CommissionEmployee Class

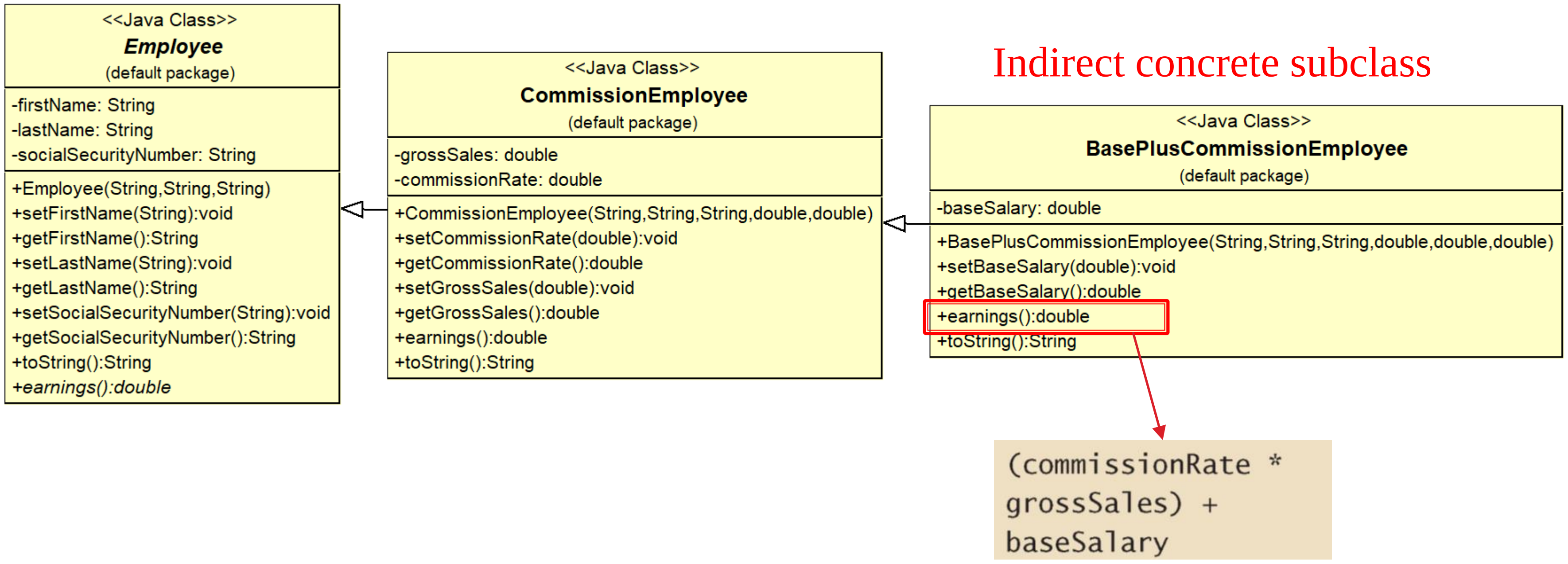
- Defines two new fields, provides the corresponding get and set methods. Provides a constructor, and overrides the `earnings` and `toString` methods.



<<Java Class>>
Employee
(default package)
-firstName: String
-lastName: String
-socialSecurityNumber: String
+Employee(String,String,String)
+setFirstName(String):void
+getFirstName():String
+setLastName(String):void
+getLastName():String
+setSocialSecurityNumber(String):void
+getSocialSecurityNumber():String
+toString():String
+earnings():double
<<Java Class>>
CommissionEmployee
(default package)
-grossSales: double
-commissionRate: double
+CommissionEmployee(String,String,String,double,double)
+setCommissionRate(double):void
+getCommissionRate():double
+setGrossSales(double):void
+getGrossSales():double
+earnings():double
+toString():String
commissionRate * grossSales

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.


Indirect concrete subclass
<<Java Class>>
Employee
(default package)
-firstName: String
-lastName: String
-socialSecurityNumber: String
+Employee(String,String,String)
+setFirstName(String):void
+getFirstName():String
+setLastName(String):void
+getLastName():String
+setSocialSecurityNumber(String):void
+getSocialSecurityNumber():String
+toString():String
+earnings():double
<<Java Class>>
CommissionEmployee
(default package)
-grossSales: double
-commissionRate: double
+CommissionEmployee(String,String,String,double,double)
+setCommissionRate(double):void
+getCommissionRate():double
+setGrossSales(double):void
+getGrossSales():double
+earnings():double
+toString():String
<<Java Class>>
BasePlusCommissionEmployee
(default package)
-baseSalary: double
+BasePlusCommissionEmployee(String,String,String,double,double,double)
+setBaseSalary(double):void
+getBaseSalary():double
+earnings():double
+toString():String
(commissionRate * grossSales) + baseSalary

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

```java
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.

`final` Methods and Static Binding

- A `final` method in a superclass cannot be overridden in a subclass. You might want to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.
- `private` methods are implicitly `final`. It’s not possible to override them in a subclass (not inherited).
- `static` methods are implicitly `final`. Non-private static methods are inherited by subclasses, but cannot be overridden (they are `final`). They are **hidden** if the subclass defines a static method with the same signature.
- A `final` method’s declaration can never change and therefore calls to `final` methods are resolved at compile time, known as **static binding**

final Methods and Static Binding

```
hello from superclass
```

```
public class TestFinalMethod {
    public static void test() {
        System.out.println("hello from superclass");
    }
    public static void main(String[] args) {
        TestFinalMethod obj = new TestFinalMethod2();
        obj.test(); // which test will be called?
    }
}

public class TestFinalMethod2 extends TestFinalMethod {
    public static void test() { // this is hiding, not overriding
        System.out.println("hello from subclass");
    }
}
```

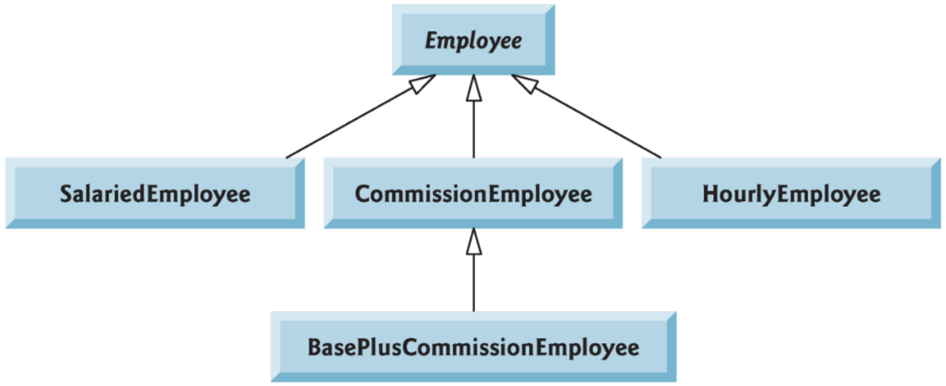
Static binding base on reference variable type, object type cannot be determined during compilation

final Classes

- A final class cannot be a superclass (cannot be extended)
  - All methods in a final class are implicitly final.
- Class String is a good example of a final class
  - If you were allowed to create a subclass of String, the subclass can override String methods in certain ways to make its object mutable. Since the subclass objects can be used wherever String objects are expected, this would break the contract that String objects are immutable.
  - Making the class final also prevents programmers from creating subclasses that might bypass security restrictions (e.g., by overriding superclass methods).

Java Interface （接口）

- We have shown that objects of related classes can be processed polymorphically by responding to the same method call in their own way (they implement common methods in their own way)



- ***Sometimes, it requires unrelated classes to implement a set of common methods. What should we do?***

Extending The Payroll System

- Suppose the company wants to use the system to calculate the money it needs to pay not only for employees but also for invoices （发票）
  - For an employee, the payment refers to the employee's earnings.
  - For an invoice, the payment refers to the total cost of the goods listed.
- In the earlier version of the system, every employee type directly or indirectly extends the abstract superclass `Employee`. The system can then manipulate different types of employee objects polymorphically.
  - Think about this: Can we make `Invoice` class extend `Employee`?
  - This is unnatural, the `Invoice` class would inherit inappropriate members (e.g., methods to obtain employee names, which have nothing to do with invoices)

Interfaces are useful in such cases.

Java Interface

- ***What is interface?*** Interfaces define and standardize the ways that objects interact with one another.

Controls on a radio serve as an interface between users and the radio’s internal components (e.g., electrical wiring)

Java Interface

- Interfaces describe a set of methods that can be called on an object, but do not provide concrete implementations for all the methods
- Different classes (radios) may implement the interfaces (controls) in different ways (e.g., using push buttons, dials, voice commands).

Java Interface

- An interface is often used when disparate (i.e., unrelated) classes need to share common methods and constants.
  - An interface is a reference type.
  - You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality.
  - A class can implement any number of interfaces (making objects polymorphic beyond the constraints of single inheritance)
  - When a class implements an interface, it has an is-a relationship with the interface data type

Java Interface

- Implementing an interface allows a class to promise certain behaviors, i.e., **forming a contract with the outside world**. This contract is enforced at build time by the compiler.

- Interfaces are useful since they capture similarity between unrelated objects without forcing a class relationship

Declaring Interfaces

- Like `public abstract` classes, interfaces are typically `public` types.
- A `public` interface must be declared in a `.java` file with the same name as the interface.

Declaring Interfaces

- An interface declaration begins with the keyword `interface` and contains only **constants** and **`abstract` methods***
  - All fields are implicitly `public`, `static` and `final`
  - All methods declared in an interface are implicitly `public abstract`

```
public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Interface names are often adjectives since interface is a way of describing what the classes can do. Class names are often nouns.

* Java 8 introduced default and static methods in interfaces, we will not consider these new features in this course.

Using Interfaces

- To use an interface, a concrete class must specify that it `implements` the interface and must implement each method in the interface with specified signature.
- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.

```
public class Invoice implements Payable {
    // must override and implement the getPaymentAmount() method
}
```

Using Interfaces

- You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface

```
Payable payableObject = new Invoice(...);
```

Interface vs. Abstract Class

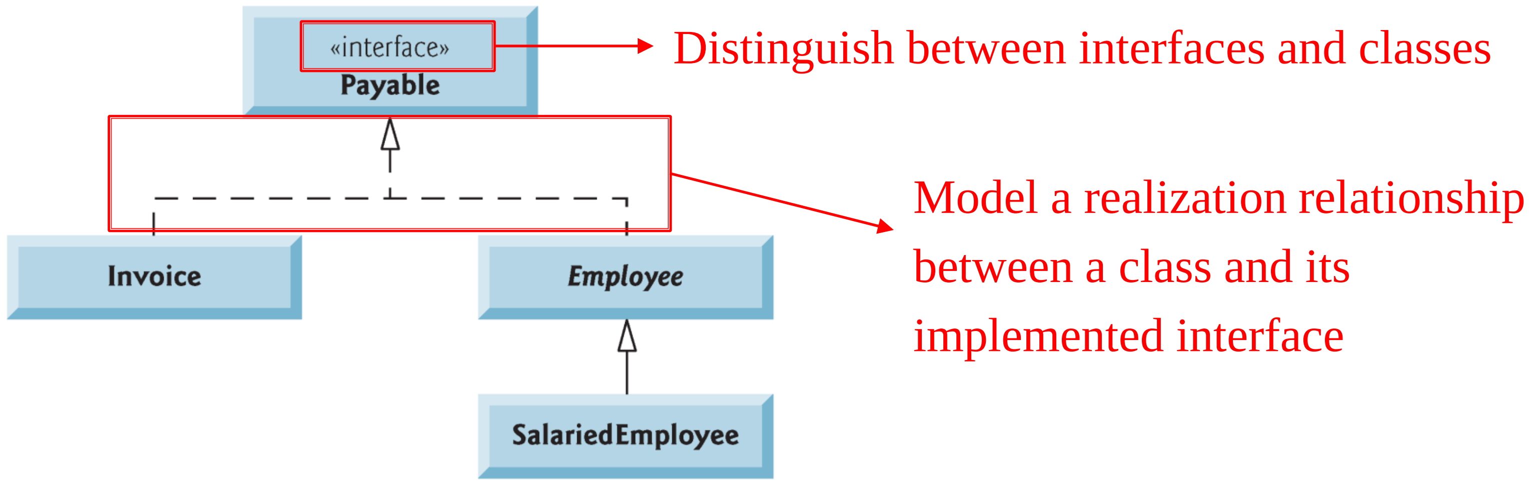
	Abstract Class	Interface
1	An abstract class can extend only one class or one abstract class	An interface can extend any number of interfaces
2	An abstract class can extend another concrete class or abstract class	An interface can only extend another interface (interfaces do not inherit from `Object`)
3	An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
4	In abstract class keyword "`abstract`" is mandatory to declare a method as an abstract	In an interface keyword "`abstract`" is optional to declare a method as an abstract
5	An abstract class can have constructors	An interface cannot have a constructor
6	An abstract class can have `protected` and `public` abstract methods	An interface can have only have `public` abstract methods
7	An abstract class can have `static`, `final` or `static final` variables with any access specifier	An interface can only have `public static final` (constant) variable

Example: Developing a Payable Hierarchy

- Extend the earlier payroll system to make it able to determine payments for both employees and invoices.
  - Classes `Invoice` and `Employee` both represent things for which the company must be able to calculate a payment amount.
  - We can make both classes implement the `Payable` interface, so a program can invoke method `getPaymentAmount` on both `Invoice` and `Employee` objects.
  - Enables the polymorphic processing of `Invoice`s and `Employee`s.

```
public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

The UML Class Diagram


«interface»
Payable
Invoice
Employee
SalariedEmployee
Distinguish between interfaces and classes
Model a realization relationship between a class and its implemented interface

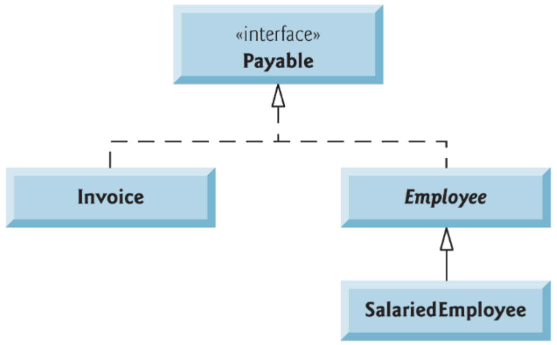
- The UML expresses the relationship between a class and an interface as realization.
  - A class is said to “realize” or implement the methods of an interface.
- A subclass inherits its superclass’s realization relationships

Interface Payable

- Interface methods are always `public` and `abstract`, so they do not need to be explicitly declared as such.
- Interfaces can have any number of methods (no implementation is allowed).
- Interfaces may also contain fields that are implicitly `final` and `static`.

```
public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Class Invoice



- Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs.
- To implement more than one interface, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class ClassName extends SuperclassName
      implements FirstInterface, SecondInterface, ...
```

```
// Invoice class that implements Payable.
public class Invoice implements Payable {
    private final String partNumber;
    private final String partDescription;
    private int quantity;
    private double pricePerItem;

    // constructor
    public Invoice(String partNumber, String partDescription, int
quantity, double pricePerItem) {
        this.partNumber = partNumber;
        this.partDescription = partDescription;
        setQuantity(quantity);
        setPricePerItem(pricePerItem);
    } // end constructor
```

The class extends `Object` (implicitly) and implements `Payable` interface

```
 // get part number
    public String getPartNumber() { return partNumber; } // should
validate
    // get description
    public String getPartDescription() { return partDescription; }
    // set quantity
    public void setQuantity(int quantity) {
        if (quantity < 0) // validate quantity
            quantity = 0;
        this.quantity = quantity;
    }
    // get quantity
    public int getQuantity() { return quantity; }
```

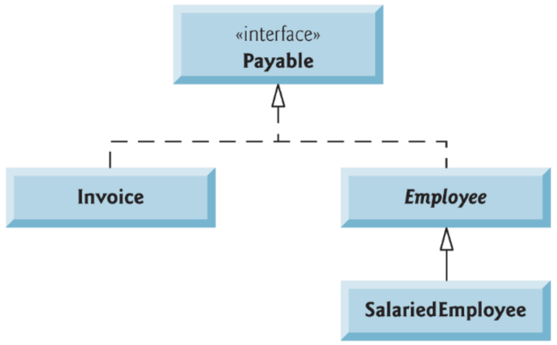
```
// set price per item
    public void setPricePerItem(double pricePerItem) {
        if (pricePerItem < 0.0) // validate pricePerItem
            pricePerItem = 0;
        this.pricePerItem = pricePerItem;
    }
    // get price per item
    public double getPricePerItem() {
        return pricePerItem;
    }

    // return String representation of Invoice object
    @Override
    public String toString() {
        return String.format("%s: %n%s: %s (%s) %n%s: %d %n%s: 
$%,.2f",
            "invoice", "part number", getPartNumber(), getPartDescription(),
            "quantity", getQuantity(), "price per item", getPricePerItem());
    }
```

```
 // method required to carry out contract with interface Payable
   @Override
   public double getPaymentAmount() {
      return getQuantity() * getPricePerItem(); // calculate total cost
   }
} // end class Invoice
```

Providing an implementation of the interface's method(s) makes this class concrete

Class Employee



- When a class implements an interface, it makes a contract with the Java compiler:
  - The class will implement each of the methods in the interface or that the class will be declared `abstract`.
  - If the latter, we do not need to declare the interface methods as `abstract` in the `abstract` class (they are already implicitly declared as such in the interface)
  - Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the contract (**the unfulfilled contract is inherited**).
  - If the subclass does not do so, it too must be declared `abstract`.

Abstract class extends `Object` (implicitly) and implements interface `Payable`

```
// Employee abstract superclass.
public abstract class Employee implements Payable{
   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;
   }

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

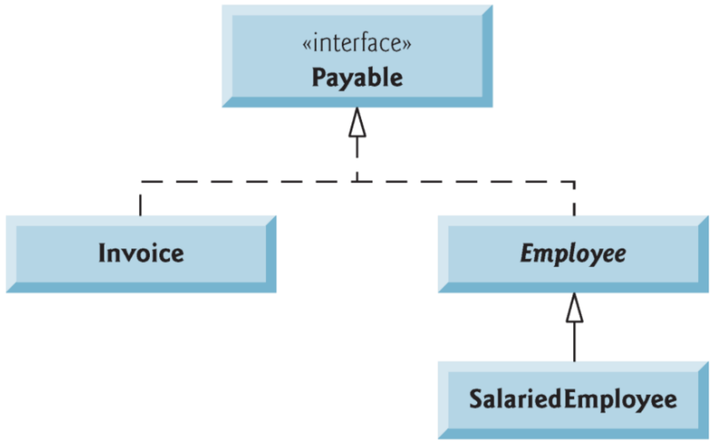
```java
// return String representation of Employee object
   @Override
   public String toString() {
      return String.format("%s %s%nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber());
   }

   // Note: We do not implement Payable method getPaymentAmount here so
   // this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

We don't implement the method, so this class needs to be declared as `abstract`.

Class SalariedEmployee

- The SalariedEmployee class that extends Employee must fulfill superclass Employee's contract to implement Payable method getPaymentAmount.



```java
// 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;
    }
```

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

   // calculate earnings; implement interface Payable method that
was
   // abstract in superclass Employee
   @Override
   public double getPaymentAmount() {
      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
```

Providing an implementation of the method to make this class concrete and instantiable

SalariedEmployee and Invoice

- Objects of a class (or its subclasses) that `implements` an interface can also be considered as objects of the interface type.
- Thus, just as we can assign the reference of a `SalariedEmployee` object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object to an interface `Payable` variable.
- `Invoice` implements `Payable`, so an `Invoice` object is also a `Payable` object, and we can assign the reference of an `Invoice` object to a `Payable` variable.

```
// Payable interface test program processing Invoices and
// Employees polymorphically.
public class PayableInterfaceTest {
   public static void main(String[] args) {
      // create four-element Payable array
      Payable[] payableObjects = new Payable[4];

      // populate array with objects that implement Payable
      payableObjects[0] = new Invoice("01234", "seat", 2, 375.00);
      payableObjects[1] = new Invoice("56789", "tire", 4, 79.95);
      payableObjects[2] = new SalariedEmployee("John", "Smith", "111-
11-1111", 800.00);
      payableObjects[3] = new SalariedEmployee("Lisa", "Barnes", "888-
88-8888", 1200.00);

      System.out.println("Invoices and Employees processed
polymorphically:");
```

An array of polymorphic objects

Assigning the references of different types of objects to the `Payable` variables

```
        // generically process each element in array payableObjects
        for (Payable currentPayable : payableObjects) {
           // output currentPayable and its appropriate payment amount
           System.out.printf("%n%s %n%s: $%,.2f%n",
   currentPayable.toString(), // could invoke implicitly
              "payment due", currentPayable.getPaymentAmount());
        }
     } // end main
  } // end class PayableInterfaceTest
```

Objects are processed polymorphically

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

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

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Common Java Interfaces

- The Java API's interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program.
  - The framework code would call certain methods defined in the interfaces and the method calls will be eventually dispatched to the methods implemented in your own classes.

Example: The `Comparable` Interface

- Java contains several comparison operators (e.g., <, >=, ==) that allow you to compare primitive values.
- However, these operators cannot be used to compare objects.
- The interface `Comparable` is used to allow objects of a class that implements the interface to be compared to one another.
- `Comparable` is commonly used for ordering objects in a collection such as an array.

```java
import java.util.Arrays;
public class Employee implements Comparable<Employee> {
    private String firstName, lastName;
    private int id;
    public Employee(String first, String last, int sid) {
        firstName = first;   lastName = last;   id = sid;
    }
    @Override
    public String toString() {
        return String.format("[%s %s ID: %d]", firstName, lastName, id);
    }
    public static void main(String[] args) {
        Employee[] employees = new Employee[3];
        employees[0] = new Employee("Jack", "Ma", 1);
        employees[1] = new Employee("Yanhong", "Li", 2);
        employees[2] = new Employee("Huateng", "Ma", 3);
        Arrays.sort(employees);
        System.out.println(Arrays.toString(employees));
    }
    @Override
    public int compareTo(Employee o) {
        return this.id - o.id;
    }
}
```

```
[[Jack Ma ID: 1], [Yanhong Li ID: 2], [Huateng Ma ID: 3]]
```