Chapter 12: Polymorphism

TAO Yida

taoyd@sustech.edu.cn

Objectives

- Polymorphism (多态)
- Abstract classes
- Interface

Polymorphism

Polymorphism

Many (Greek) Form

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

Light-morph jaguar

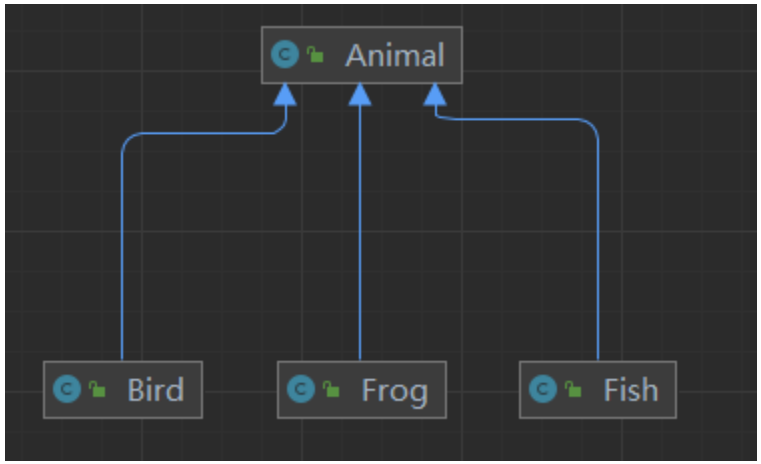
Dark-morph jaguar

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

Motivating Example

- **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`.




```
System.out.println("Bird flies");
```

```
System.out.println("Frog jumps");
```

```
System.out.println("Fish swims");
```

Motivating Example

A zookeeper's daily job is to make every animal move.

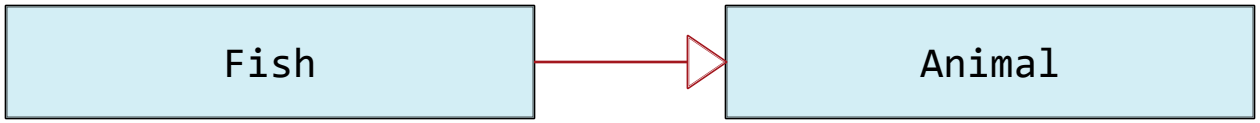
```
public class Zookeeper {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.move();

        Fish fish = new Fish();
        fish.move();

        Frog frog = new Frog();
        frog.move();
    }
}
```

```
Bird flies
Fish swims
Frog jumps
```

What if the zoo has 100 different types of animals?

Polymorphic Behavior

- Earlier, when we write programs, we let super class variables refer to superclass objects and subclass variables refer to subclass objects




```
Animal animal = new Animal();
Fish fish = new Fish();
```

Such assignments are natural

Polymorphic Behavior

- In Java, we can also let a superclass variable refer to a subclass object (the most common use of polymorphism)

```
Animal animal = new Fish();
```

父类引用指向子类对象

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

```
Fish fish = new Animal();
```

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

Polymorphic Behavior

- In Java, we can also let a superclass variable refer to a subclass object (the most common use of polymorphism)

```
Animal  animal    =    new  Fish();
```

Type recognized by the compiler. Use in the compilation time.

Type recognized by JVM. Use in the execution time.

Polymorphic Behavior

- Then the question comes…

```
Animal animal = new Fish();
animal.move();
```

多态特性：实例方法调用是基于运行时的实际类型(=右边)的动态调用，而非变量的声明类型(=左边)。

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

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

This process is called **dynamic binding (动态绑定)**. Binding means "associating method calls to the appropriate method body".

Polymorphic Behavior

- Suppose the `Fish` class adds a new method `sleepEyesOpen()`. Can the code compile?

```
Animal animal = new Fish();
animal.sleepEyesOpen();
```

  - Compilation error: Cannot resolve method 'sleepEyesOpen' in 'Animal'.
  - Compiler only knows that animal's type is Animal, so it checks whether the Animal class has the method sleepEyesOpen(). If not, a compilation error occurs.

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

Polymorphic Behavior

- To avoid such compilation errors, the superclass object's reference must be downcast (向下转型) to a subclass type explicitly.

```
Animal animal = new Fish();

if(animal instanceof Fish){
    // downcasting
    Fish fish = (Fish) animal;
    fish.sleepEyesOpen();

}
```

Use the **`instanceof`** operator to ensure that such a cast is performed only if the object is a subclass object.

```
Animal animal = new Bird();

Fish fish = (Fish) animal;
fish.sleepEyesOpen();
```

At runtime, if the object to which the reference refers is not a subclass object, a ClassCastException will occur

Polymorphic Behavior

```
Animal animal = new Fish();
animal.move();
animal.sleepEyesOpen();
```

多态特性：

- 父类引用指向子类对象
- 父类引用可以调用父类所有成员（需遵守访问权限）
- 父类引用不能调用子类中特有成员
- 方法调用的最终运行效果看子类里的具体实现（若子类没有重写此方法，则依次向上层父类查找）

Back to the Motivating Example

```
public class Zookeeper {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.move();

        Fish fish = new Fish();
        fish.move();

        Frog frog = new Frog();
        frog.move();
    }
}
```

No polymorphism

```
public class Zookeeper {
    public static void main(String[] args) {
        Animal bird = new Bird();
        bird.move();

        Animal fish = new Fish();
        fish.move();

        Animal frog = new Frog();
        frog.move();

    }
}
```

With polymorphism

```
Bird flies
Fish swims
Frog jumps
```

Polymorphism Use Cases

```
public class Zookeeper {
    3 usages
    public static void manage(Animal animal){
        animal.move();
    }

    public static void main(String[] args) {
        Animal bird = new Bird();
        Animal fish = new Fish();
        Animal frog = new Frog();

        manage(bird);
        manage(fish);
        manage(frog);
    }
}
```

← Different behaviors during runtime

多态应用：方法声明的形参类型为父类类型，可以使用子类的对象作为实参调用该方法

```
Bird flies
Fish swims
Frog jumps
```

Without polymorphism, we need to define many overloaded manage methods: `manage(Bird bird)`, `manage(Fish fish)`, `manage(Frog frog)`…

Polymorphism Use Cases

```
Animal bird = new Bird();
Animal fish = new Fish();
Animal frog = new Frog();

ArrayList<Animal> list = new ArrayList<>();
list.add(bird);
list.add(fish);
list.add(frog);

for(Animal animal: list){
    animal.move();
}
```

多态应用：多态数组

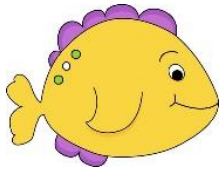
← Different behaviors during runtime

```
Bird flies
Fish swims
Frog jumps
```

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 execution results—hence the term polymorphism.

- **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`



. . .

Polymorphism in OOP

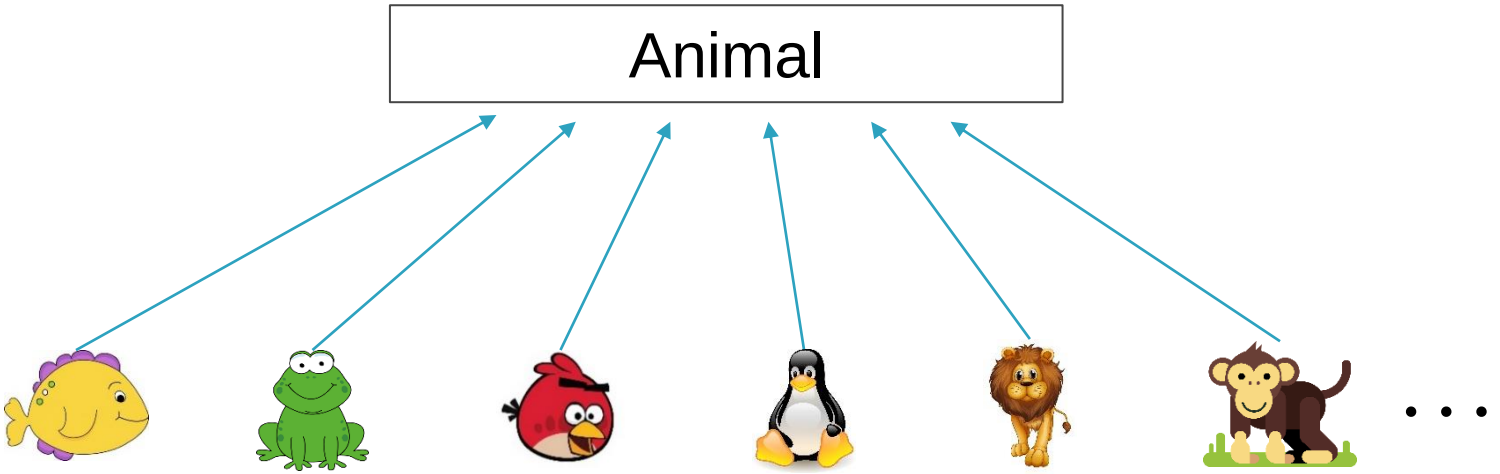
- Polymorphism enables you to write programs to process objects that share the same superclass as if they're all objects of the superclass.
- With polymorphism, we can design and implement *extensible* systems (可扩展的)
  - New classes can be added with little or no modification to the general portions 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).

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.

  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.

  We can use abstract classes and abstract methods



Objectives

- Polymorphism
- Abstract classes
- Interface

Concrete classes & Concrete methods

- 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**" (具体类)
- Concrete classes can be used to instantiate objects (i.e., we can `new ConcreteClass(…)`)
- Methods with implementations (method body) are called "concrete methods"

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 and mostly for design purposes
- They are called "**abstract classes**", CANNOT be used to instantiate objects (cannot `new AbstractClass(…)`)

Declaring Abstract Classes

- You make a class abstract by declaring it with keyword `abstract`.
- An abstract class normally contains one or more abstract methods; However, we can also declare an abstract class with no abstract method
- Abstract methods are declared with the keyword `abstract` and provides no implementations.

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

→ Be careful, no brackets {}

Abstract Method

- Abstract methods have no implementations because the abstract classes are too general
- Abstract methods specify the common interfaces (method signature) that subclasses need to implement
- If a class includes any abstract method, then the class itself must be declared abstract (even if the class also has concrete methods)

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

Abstract methods cannot be private, final, or static

- 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 cannot be `static` or `final`

`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.
- A `final` method's declaration can never change and therefore calls to `final` methods are resolved at compile time, known as **static binding (静态绑定)**
- `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 **hidden** if the subclass defines a static method with the same signature.

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");
    }
}
```

When we hides a static method, we do not get the benefit of run-time polymorphism.

Static binding is decided by compiler and is based on variable type (=左边) instead of actual object type (=右边)

`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

```
public final class String
extends Object
```

  - 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).

Using Abstract Classes

- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

- 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. (Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract)

Using Abstract Classes

- 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`
  - `moveAnimal(Animal a) { a.move() }` When called, such a method can receive an object of any concrete class that directly or indirectly extends the abstract superclass `Animal`.
- Such practice is commonly adopted to manipulate objects polymorphically.

Using Abstract Classes

- An abstract class is essentially a class; it can also have fields, constructors, and concrete methods

```java
public abstract class Animal {
    2 usages
    int position_x;
    2 usages
    int position_y;

    1 usage
    Animal(){
        position_x = 10;
        position_y = 10;
    }

    1 usage  1 implementation
    public abstract void move();

    1 usage
    public void sleep(){
        System.out.println("zzz...");
    }
}
```

```java
public class Fish extends Animal {
    1 usage
    @Override
    public void move(){
        System.out.println("Fish swims");
    }
}
```

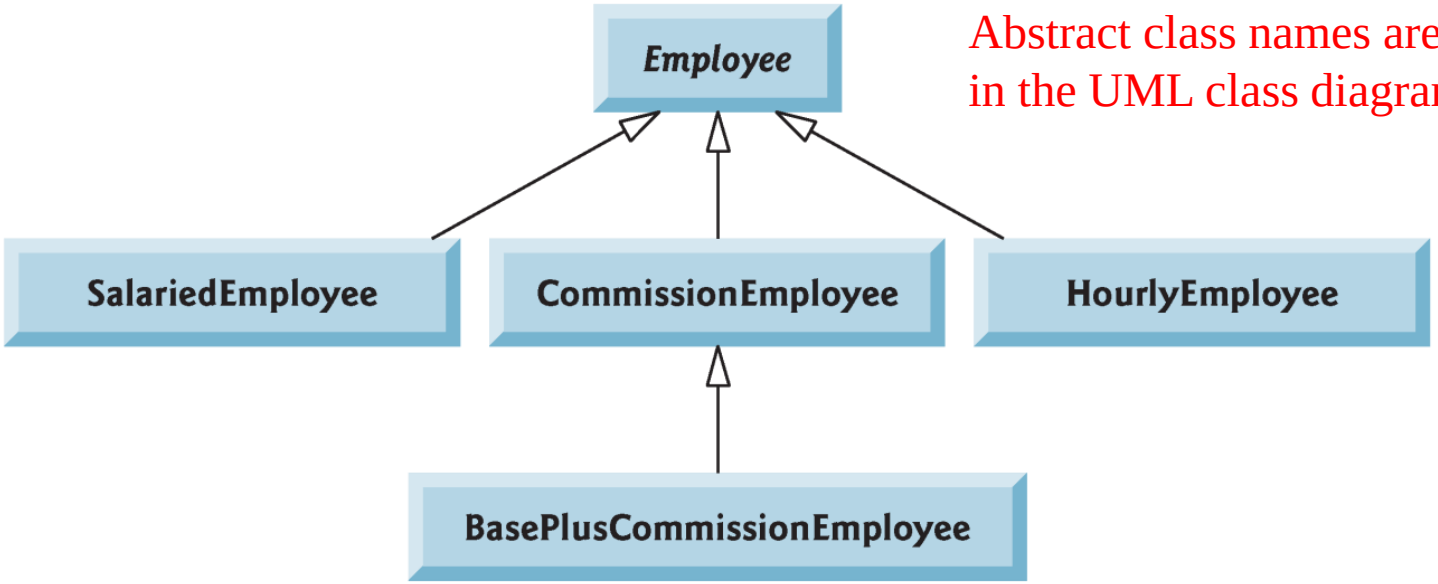
Constructors cannot be declared `abstract` (constructors are not inherited)

```java
public static void main(String[] args) {
    Animal fish = new Fish();

    System.out.println(fish.position_x); // 10
    System.out.println(fish.position_y); // 10

    fish.move(); // "Fish swims"
    fish.sleep(); // "zzz..."
}
```

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.
- The company wants to write a Java application that performs its payroll calculations polymorphically.

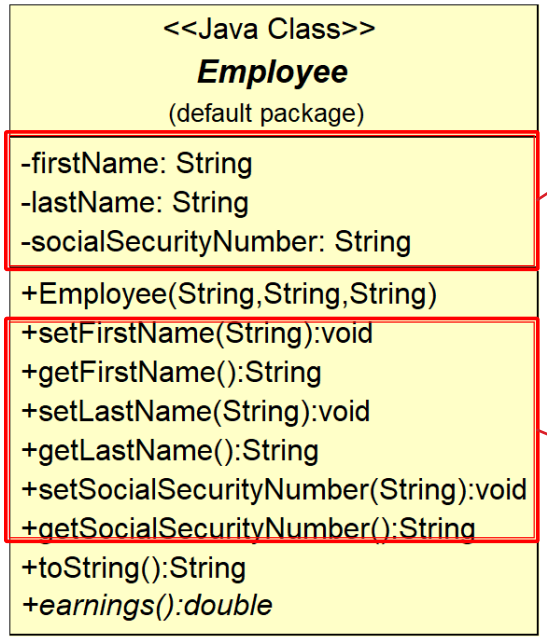
The Design: Main Classes



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.

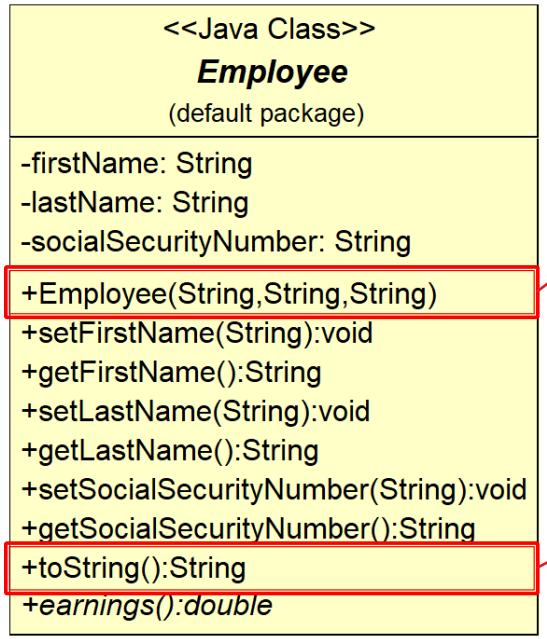


Each employee has a first name, a last name and a social security number. This applies to all employee types.

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.

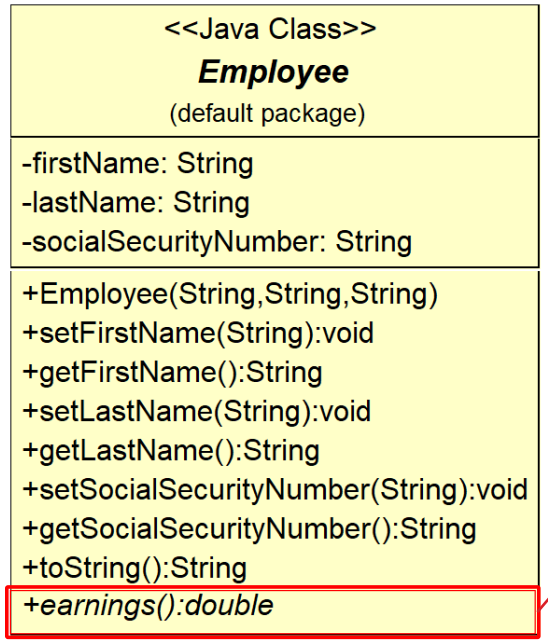


A constructor for initializing the three fields

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.

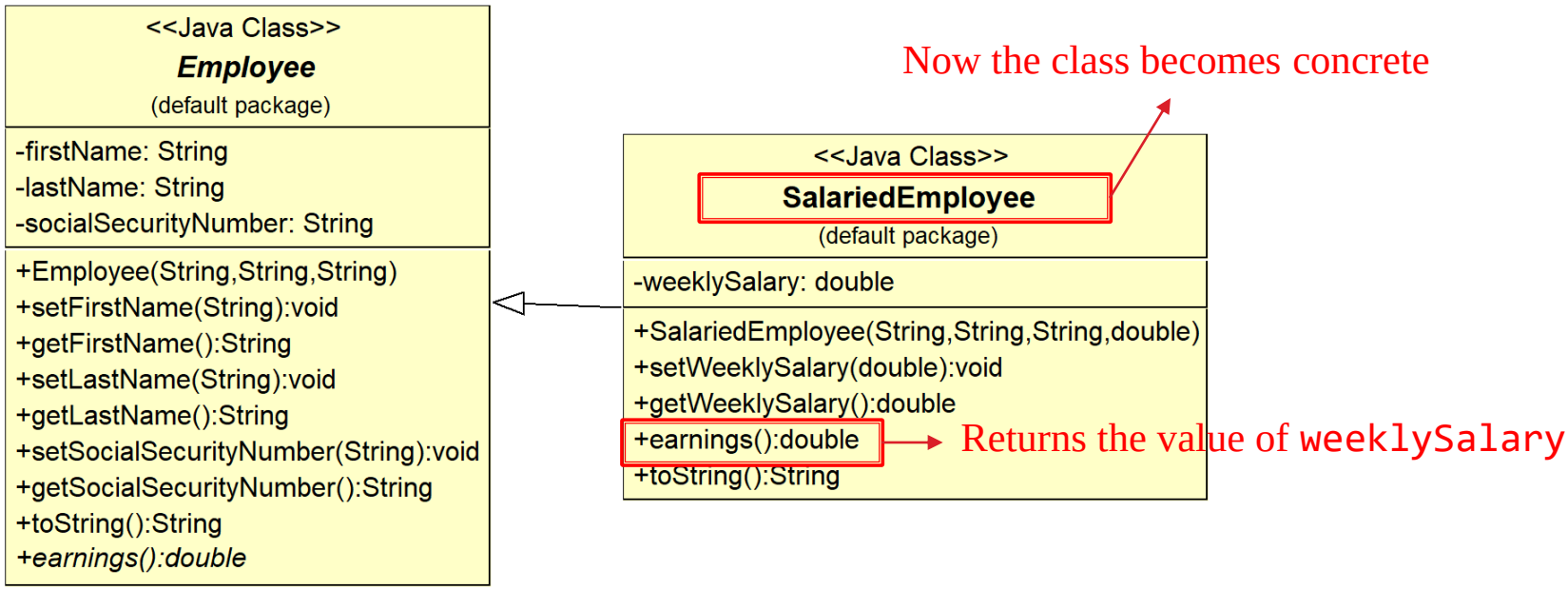


Abstract method names are italicized

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

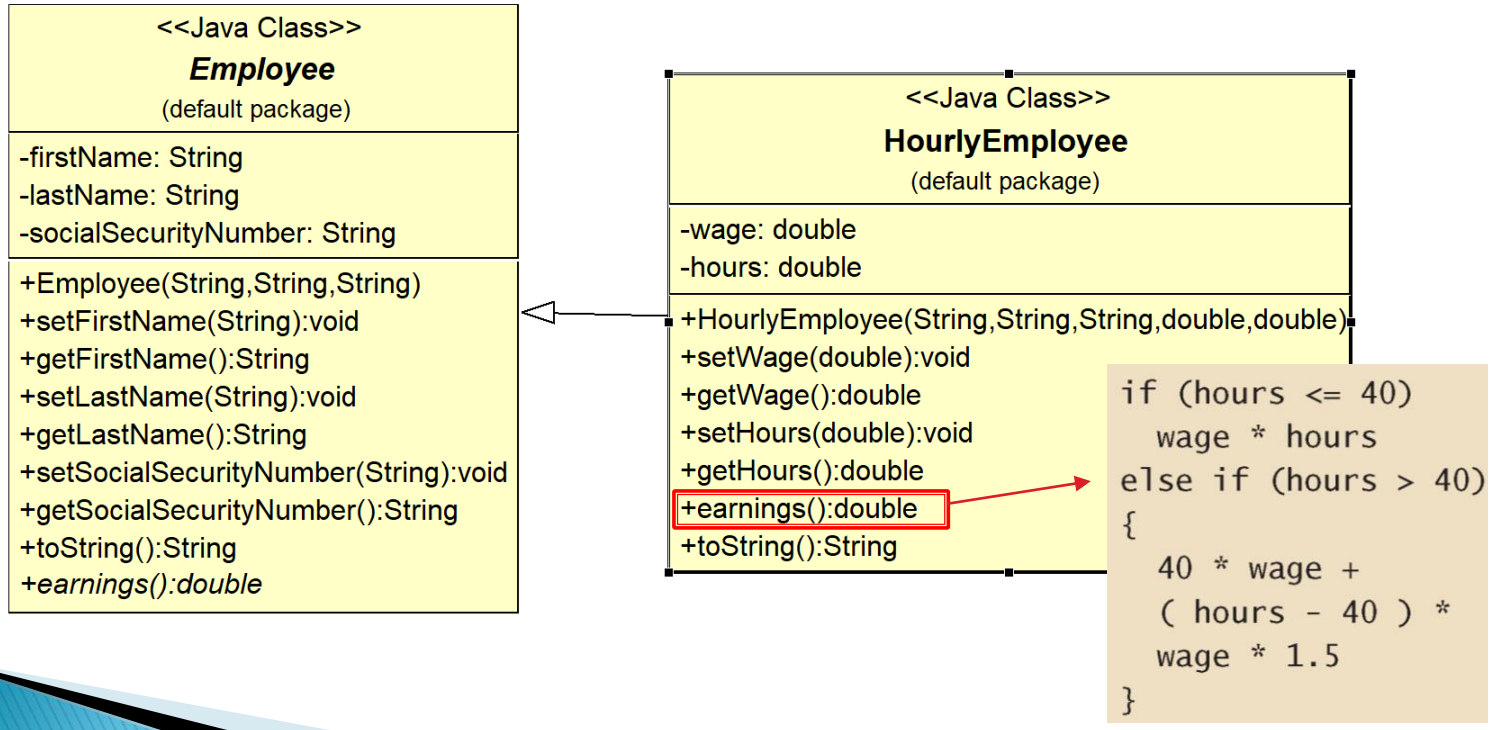
The SalariedEmployee Class

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



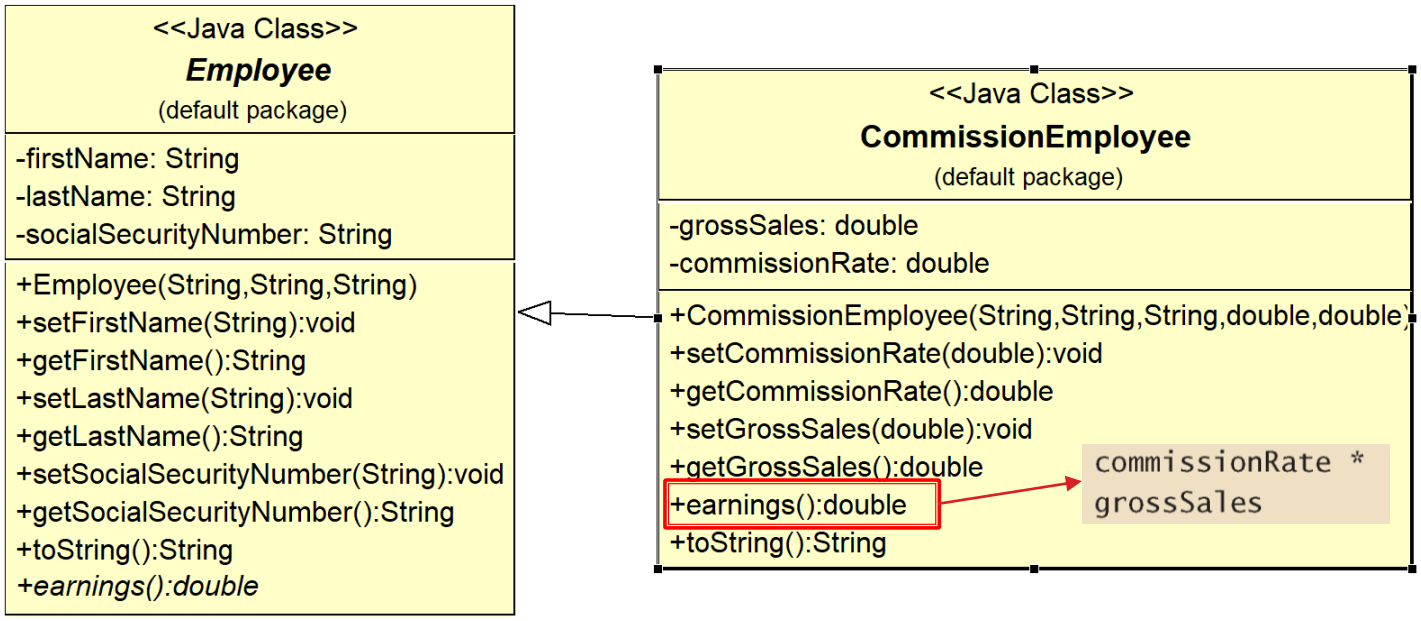
The HourlyEmployee Class

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



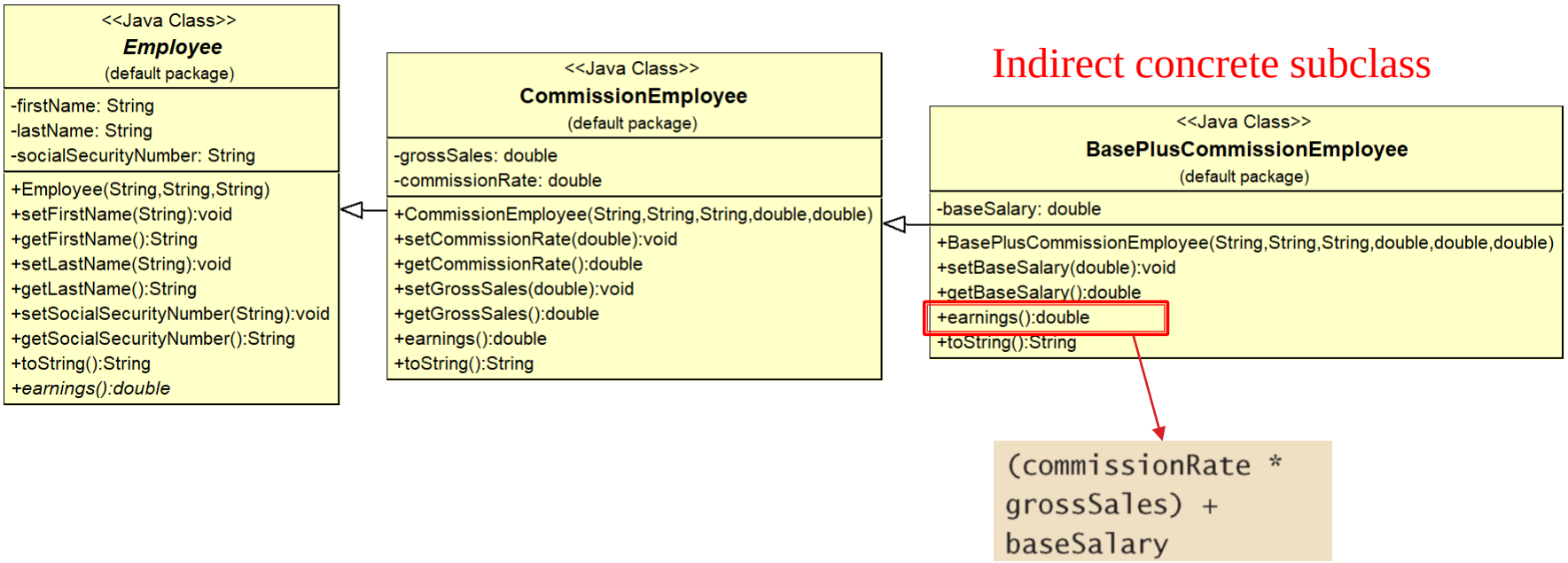
The CommissionEmployee Class

- 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

```
public abstract class Employee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName
```

Although abstract classes cannot be used to instantiate objects, they can have constructors, which can be leveraged by subclasses

```
// return first name
public String getFirstName()
{
   return firstName;
} // end method getFirstName

// set last name
public void setLastName( String last )
{
   lastName = last; // should validate
} // end method setLastName

// return last name
public String getLastName()
{
   return lastName;
} // end method getLastName

// set social security number
public void setSocialSecurityNumber( String ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end method setSocialSecurityNumber
```


```
    // return social security number
    public String getSocialSecurityNumber()
    {
       return socialSecurityNumber;
    } // end method getSocialSecurityNumber

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

    // abstract method overridden by concrete subclasses
    public abstract double earnings(); // no implementation here
} // end abstract class Employee
```


The SalariedEmployee Class

```
public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary
```

Initialize private fields that are not inherited


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

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

    // 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 method toString
} // end class SalariedEmployee
```

Code reuse, good practice

The HourlyEmployee Class

```
public class HourlyEmployee extends Employee
{
   private double wage; // wage per hour
   private double hours; // hours worked for week

   // five-argument constructor
   public HourlyEmployee( String first, String last, String ssn,
      double hourlyWage, double hoursWorked )
   {
      super( first, last, ssn );
      setWage( hourlyWage ); // validate hourly wage
      setHours( hoursWorked ); // validate hours worked
   } // end five-argument HourlyEmployee constructor

   // set wage
   public void setWage( double hourlyWage )
   {
      wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
   } // end method setWage
```


```
// return wage
public double getWage()
{
   return wage;
} // end method getWage

// set hours worked
public void setHours( double hoursWorked )
{
   hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
      hoursWorked : 0.0;
} // end method setHours

// return hours worked
public double getHours()
{
   return hours;
} // end method getHours
```



```
   // 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;
   } // end method earnings

   // 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 method toString
} // end class HourlyEmployee
```

Code reuse, good practice

The CommissionEmployee Class

```
public class CommissionEmployee extends Employee
{
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      super( first, last, ssn );
      setGrossSales( sales );
      setCommissionRate( rate );
   } // end five-argument CommissionEmployee constructor

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


```
// return commission rate
public double getCommissionRate()
{
   return commissionRate;
} // end method getCommissionRate

// set gross sales amount
public void setGrossSales( double sales )
{
   grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end method setGrossSales

// return gross sales amount
public double getGrossSales()
{
   return grossSales;
} // end method getGrossSales

// calculate earnings; override abstract method earnings in Employee
@Override
public double earnings()
{
   return getCommissionRate() * getGrossSales();
} // end method earnings
```



```
   // 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() );
   } // end method toString
} // end class CommissionEmployee
```

Code reuse good practice

The BasePlusCommissionEmployee Class

```
public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
   } // end method setBaseSalary
```



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

    // calculate earnings; override method earnings in CommissionEmployee
    @Override
    public double earnings()
    {
       return getBaseSalary() + super.earnings();
    } // end method 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() );
    } // end method toString
} // end class BasePlusCommissionEmployee
```


Putting Things Together

```
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

Assigning a superclass object's reference to a superclass variable is **natural**.

Assigning a subclass object's reference to a subclass variable is **natural**.

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

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

System.out.printf( "%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() );
```


```
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
```


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.println( "Employees processed polymorphically:\n" );

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

   System.out.println( currentEmployee.earnings() );
}
```

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

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.println( "Employees processed polymorphically:\n" );

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

   System.out.println( currentEmployee.earnings() );
}
```

What if for the current pay period, the company has decided to reward BasePlusCommission employees by adding 10% to their base salaries?

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 getBaseSalary() and setBaseSalary() methods cannot be invoked (Superclass reference can be used to invoke only methods of the superclass)


```
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
```


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", i,
      employees[ j ].getClass().getName() );
```

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)

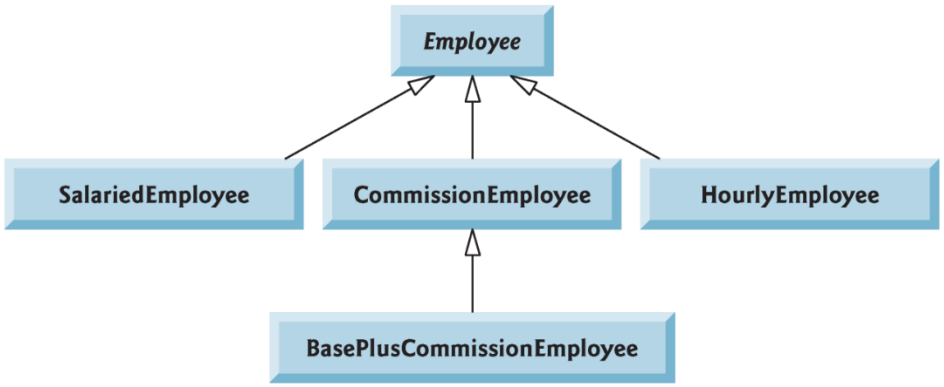
```
Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Objectives

- Polymorphism
- Abstract classes
- Interface

Extending The Payroll System

- 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)



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.

- ***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
  - For an employee, the payment refers to the employee's earnings.
  - For an invoice (发票), the payment refers to the total cost of the goods listed.

- 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 we've learned so far: Objects define their interaction with the outside world through the methods that they expose

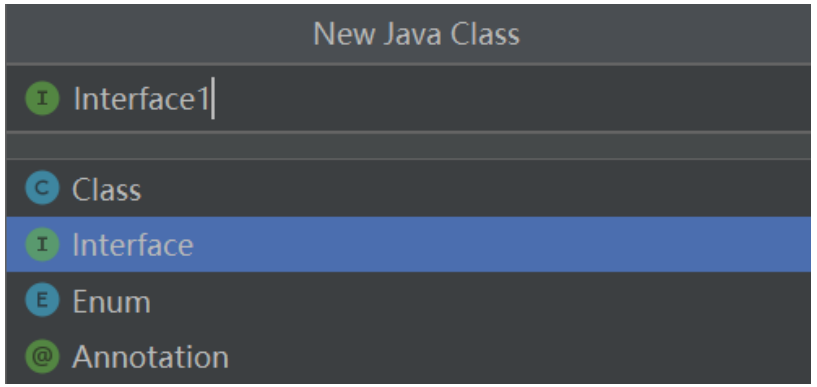
- Interfaces describe a set of methods (signatures) that can be called on an object, but do not provide concrete implementations for all the methods (like a completely "abstract class")

- To access the interface methods, the interface must be "implemented" by another class with the `implements` keyword

Declaring Interfaces

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

```
public interface Interface1 {
}
```



Declaring Interfaces

- An interface can `extend` other interfaces, just as a class extends another class.
- Whereas a class can extend only one other class, an interface can extend any number of interfaces (separated by comma)
- An interface cannot `extend` a class, and cannot `implement` other interfaces
  - This would cause a conflict with the fact that interfaces are "abstract"

```
public interface Interface3 extends Interface1, Interface2{
}
```


The Interface Body - Methods

- The interface body can contain abstract, default, and static methods.
  - Default methods and static methods in interfaces are allowed since Java 8
  - All abstract, default, and static methods in an interface are implicitly public, so you can omit the `public` modifier.
- Default methods are defined with the `default` modifier, and static methods with the `static` keyword; both default and static methods should have concrete implementations.
- Methods in an interface are implicitly `abstract` if they are not static or default

```
public interface Interface1 {
    default void foo(){
        System.out.println("default method");
    }

    static void bar(){
        System.out.println("static method");
    }

    void baz();
}
```


The Interface Body - Methods

- Static methods
  - Typically used for utility methods
  - Can be invoked by InterfaceName.methodName(…)

- Default methods
  - Classes that implement an interface can invoke the default methods in the interface
  - Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

The Interface Body

- An interface can contain constant declarations.
- All constant fields (values) defined in an interface are implicitly `public`, `static`, and `final`. You can omit these modifiers.
- An interface in Java doesn't have a constructor (as all fields are constants, there is no need to initialize them through the constructor)

```
public interface Interface2 {

    int var = 10;

    void baz();

}
```


The Interface Body

Java 7 or earlier

- Constant variables
- Abstract methods

Java 8

- Constant variables
- Abstract methods
- Default methods
- Static methods

Java 9 or later

- Constant variables
- Abstract methods
- Default methods
- Static methods
- Private methods
- Private static methods

Implementing an Interface

```
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.

- To use an interface, a concrete class must specify that it `implements` the interface and must implement ALL abstract methods in the interface with specified signature. If not, the class must be abstract.

```
public class Invoice implements Payable {

    // must override and implement the getPaymentAmount() method

}
```


Implementing an Interface

- A class can inherit from only one superclass, but can implement as many interfaces as it needs.

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

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```


Using an Interface as a Type

- An interface is a reference type.
- You can use interface names anywhere you can use any other data type name.
- We cannot instantiate an interface directly
- 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

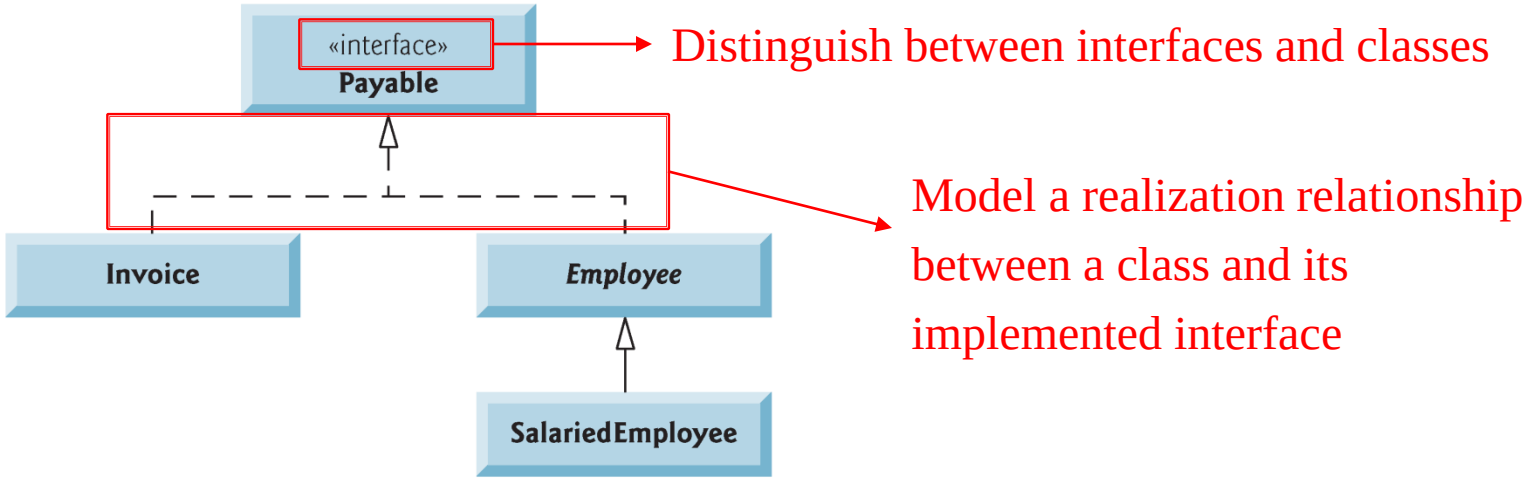
```
Payable payableObject = new Invoice(...);
```


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
```

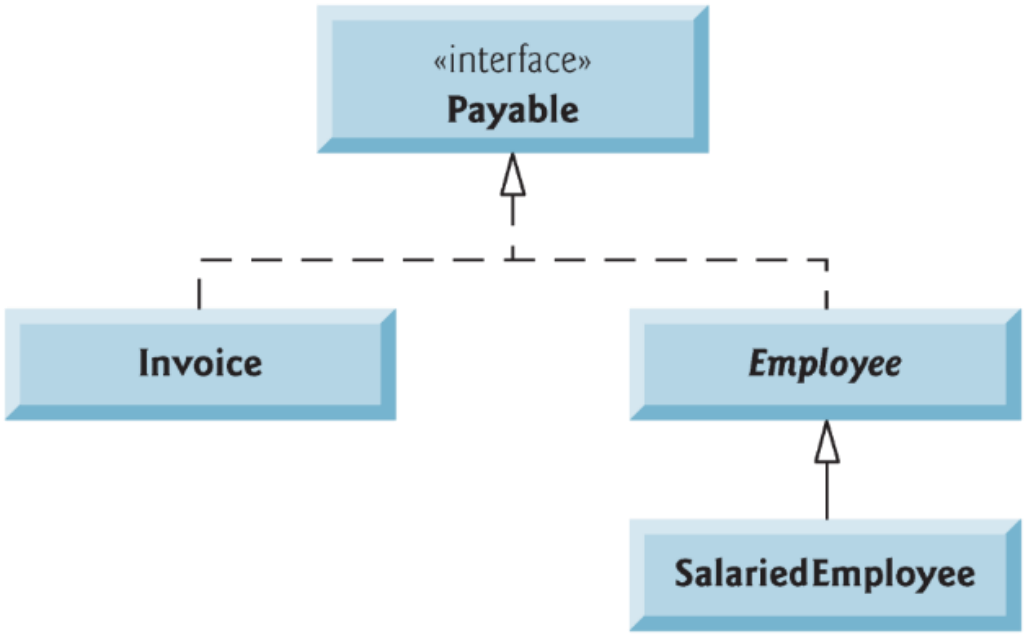

The UML Class Diagram



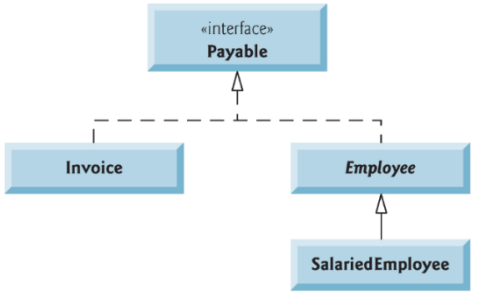
- The UML expresses the relationship between a class and an interface as realization.
  - A class is said to "realize" or implement the methods of an interface.
- A subclass inherits its superclass's realization relationships

Interface Payable

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



Class Invoice



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

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

   // four-argument constructor
   public Invoice( String part, String description, int count,
      double price )
   {
      partNumber = part;
      partDescription = description;
      setQuantity( count ); // validate and store quantity
      setPricePerItem( price ); // validate and store price per item
   } // end four-argument Invoice constructor
```

```
// set part number
public void setPartNumber( String part )
{
   partNumber = part; // should validate
} // end method setPartNumber

// get part number
public String getPartNumber()
{
   return partNumber;
} // end method getPartNumber

// set description
public void setPartDescription( String description )
{
   partDescription = description; // should validate
} // end method setPartDescription

// get description
public String getPartDescription()
{
   return partDescription;
} // end method getPartDescription
```

```
// set quantity
public void setQuantity( int count )
{
   quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
} // end method setQuantity

// get quantity
public int getQuantity()
{
   return quantity;
} // end method getQuantity

// set price per item
public void setPricePerItem( double price )
{
   pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
} // end method setPricePerItem

// get price per item
public double getPricePerItem()
{
   return pricePerItem;
} // end method getPricePerItem
```

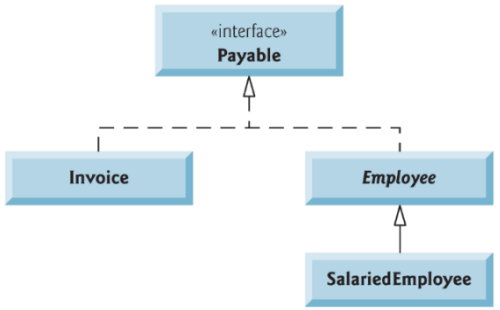
```
    // return String representation of Invoice object
    @Override
    public String toString()
    {
       return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
          "invoice", "part number", getPartNumber(), getPartDescription(),
          "quantity", getQuantity(), "price per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

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

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

Class Employee

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

```
public abstract class Employee implements Payable
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName
```



```
// return first name
public String getFirstName()
{
   return firstName;
} // end method getFirstName

// set last name
public void setLastName( String last )
{
   lastName = last; // should validate
} // end method setLastName

// return last name
public String getLastName()
{
   return lastName;
} // end method getLastName

// set social security number
public void setSocialSecurityNumber( String ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end method setSocialSecurityNumber
```

```
   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

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

   // 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`.

Implementing an Interface

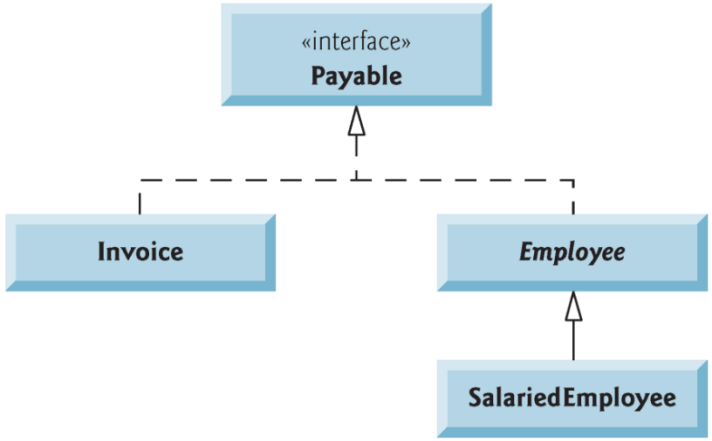
- When a class implements an interface, it makes a contract (协议) with the Java compiler:
  - The class will implement each of the methods in the interface; If the subclass does not do so, it too must be declared `abstract`.
  - Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the contract (**the unfulfilled contract is inherited**).

```
class Invoice implements Payable{

      double getPaymentAmont(){
          ……..
      }
}
```

Interface is a “capability”;
If a class claims to have a capability (implements an interface), it must override all its abstract methods before the compiler admits that it indeed has that capability

Class SalariedEmployee

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



```
public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary
```

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

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

    // 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 method toString
} // 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.
- ```
  Payable payableObject = new SalariedEmployee(...);
  ```
- `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 payableObject = new Invoice(...);
    ```

```
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:\n" );
```

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( "%s \n%s: $%,.2f\n\n",
            currentPayable.toString(),
            "payment due", currentPayable.getPaymentAmount() );
      } // end for
   } // 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
```

Implementing Interface vs. Extending class

- Inheritance (extending a class)
  - Provide code reusability (subclasses reuse superclass's features)
  - Single inheritance

- Interface
  - Provides abstraction (used for design purposes, cannot be instantiated)
  - Multiple inheritance
  - Useful and more flexible since they capture similarity between unrelated objects without forcing a class relationship

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 cannot extend classes
3	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
4	An abstract class can have constructors	An interface cannot have a constructor
5	An abstract class can have `protected` and `public` abstract methods	An interface can only have `public` abstract methods
6	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