Chapter 11: Polymorphism

TAO Yida

taoyd@sustech.edu.cn

Objectives

- Polymorphism (多态)
- Abstract classes (抽象类)

Polymorphism

Polymorphism

Many (Greek) Form

- The word **polymorphism** is used in various disciplines to describe situations where something occurs in several different forms
- Biology example: About 6% of the South American population of jaguars are dark-morph jaguars.

Light-morph jaguar

Dark-morph jaguar

Polymorphism

- The word **polymorphism** is used in various disciplines to describe situations where something occurs in several different forms
- Biology example: About 6% of the South American population of jaguars are dark-morph jaguars.

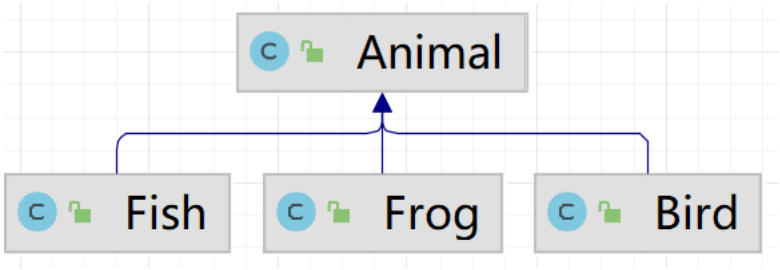
- In Java or OOP, **polymorphism** is the ability of an object to take on many forms.

Motivating Example

- **Example:** Suppose we create a program that simulates the movement of several types of animals for a zoo application. 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`.

```
@Override
public void move(){
    System.out.println("Fish swims");
}
```




```
@Override
public void move(){
    System.out.println("Frog jumps");
}
```

```
@Override
public final void move(){
    System.out.println("Bird flies");
}
```

Motivating Example

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

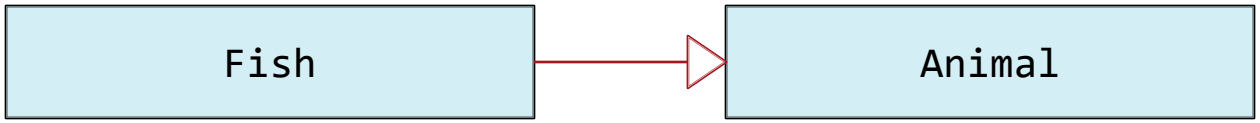
```
public class Zookeeper {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.move();

        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 superclass 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(actual type, =右边)的动态调用，而非变量的声明类型(Declared type, =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)**, discussed later.

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.

Polymorphic Behavior

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

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

- 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 (declared 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（基于声明类型）
- 方法调用的最终运行效果看子类里的具体实现 （基于实际类型）
  （若子类没有重写此方法，则依次向上层父类查找）

Polymorphism Use Cases

```
public class Zookeeper {

    public static void main(String[] args) {
        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 Use Cases

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

    public static void main(String[] args) {
        manage(new Bird());
        manage(new Fish());
        manage(new Frog());
    }
}
```

← Different behaviors during runtime

Bird flies
Fish swims
Frog jumps

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

Without polymorphism, we need to define many overloaded manage methods:

```
manage(Bird bird),
manage(Fish fish),
manage(Frog frog)…
```

Method Binding

Method binding refers to the process of associating a method call with its implementation

- **Dynamic binding**: the method that will be executed is determined at runtime.
- **Static binding**: the method that will be executed is determined at compile-time.

Dynamic Binding

- Also known as late binding or runtime polymorphism.
- The method to be executed is determined at runtime, based on the actual type of the object, not the reference type (declared type)

```
Animal animal = new Fish();

animal.move();
```

Static Binding

- Also known as early binding or compile-time binding.
- The compiler determines the method to be called based on the declared type of the reference variable.

```
static double    max(double a, double b)
static float     max(float a, float b)
static int       max(int a, int b)
static long      max(long a, long b)
```

```
int a = 2;
int b = 3;
Math.max(a, b);
```

Which version of `max`?

Method overloading is an example of static binding

Static Binding

- `private` methods: such methods are not inherited, can be determined at compile time

- `static` methods: such methods are bound to its class, can be determined at compile time
  - 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.

Method Hiding Example

```
public class Animal {
    public static void eat(){
        System.out.println("Animal eats");
    }
}

public class Cat extends Animal{
    public static void eat(){
        System.out.println("Cat eats");
    }

    public static void main(String[] args) {
        Animal x = new Cat();
        x.eat(); // same as Animal.eat()
    }
}
```

Static binding: static methods are bonded to the declared type during compile time

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 determined at compile time
- `private` methods are implicitly `final`.
- `static` methods are implicitly `final`.

`final` Classes

- A `final` class cannot be a superclass (cannot be extended)
  - All methods in a `final` class are implicitly `final`.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions (e.g., by overriding superclass methods).

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

```
public final class System
extends Object
```

Type Cast and `instanceof`

```
public class Zookeeper {

    public static void checkIndividual(Animal animal){
        if(animal instanceof Bird){
            Bird bird = (Bird) animal;
            bird.cleanFeathers();
        } else if(animal instanceof Fish){
            Fish fish = (Fish)animal;
            fish.sleepEyesOpen();
        }
    }

    public static void main(String[] args) {
        checkIndividual(new Bird());
        checkIndividual(new Fish());
    }
}
```

Implicit casting:

`new Bird()` is assigned to `animal`

Explicit casting:

`animal` casts to `bird`, allowed only when `animal` is an instance of `Bird`

`bird` and `animal` refers to the same object (no new object is created with casting)

Motivation for Polymorphism

- Polymorphism enables clients to use the same way to interact with objects of multiple types.

```
public class Zookeeper {

    public static void main(String[] args) {
        ArrayList<Animal> list = new ArrayList<>();
        list.add(new Bird());
        list.add(new Fish());
        list.add(new Frog());

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

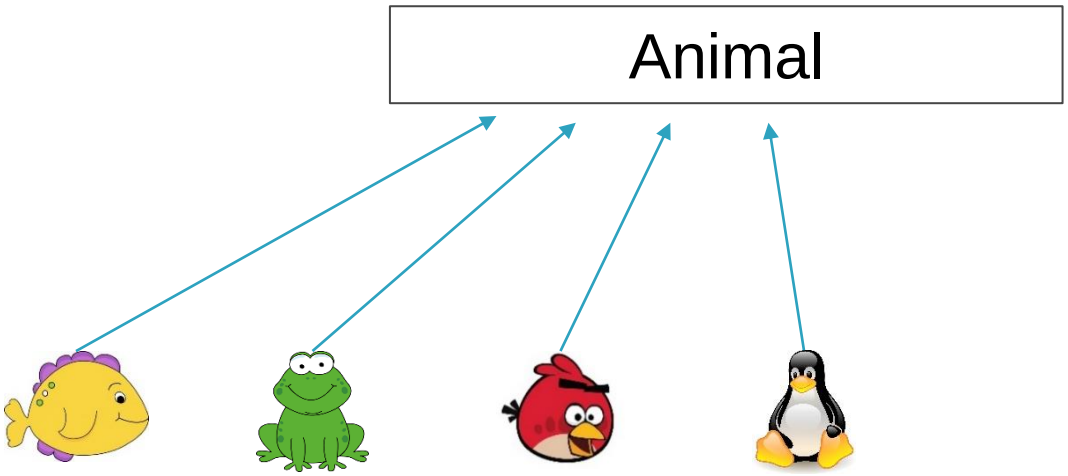
```
public class Zookeeper {

    public static void manage(Animal animal){
        animal.move();
    }

    public static void main(String[] args) {
        manage(new Bird());
        manage(new Fish());
        manage(new Frog());
    }

}
```

Subclasses can be used wherever the superclass is expected.

Motivation for Polymorphism

- 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).

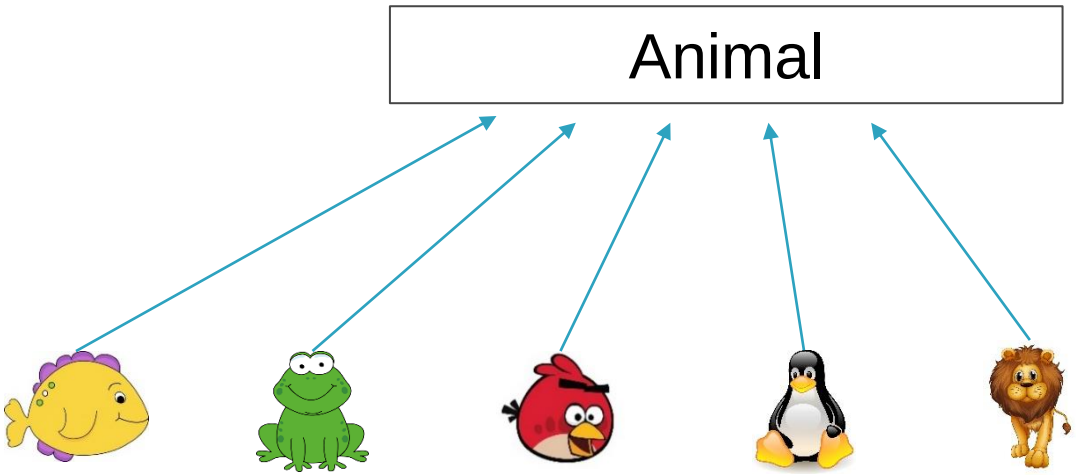
```
manage(new Penguin());
```



Motivation for Polymorphism

- 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).

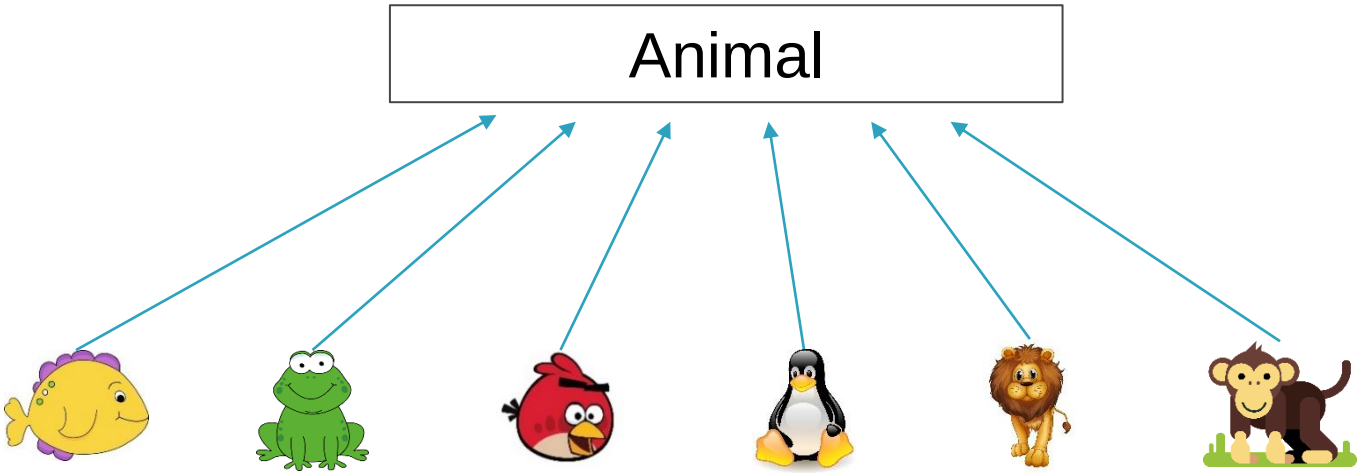
```
manage(new Lion());
```



Motivation for Polymorphism

- 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).

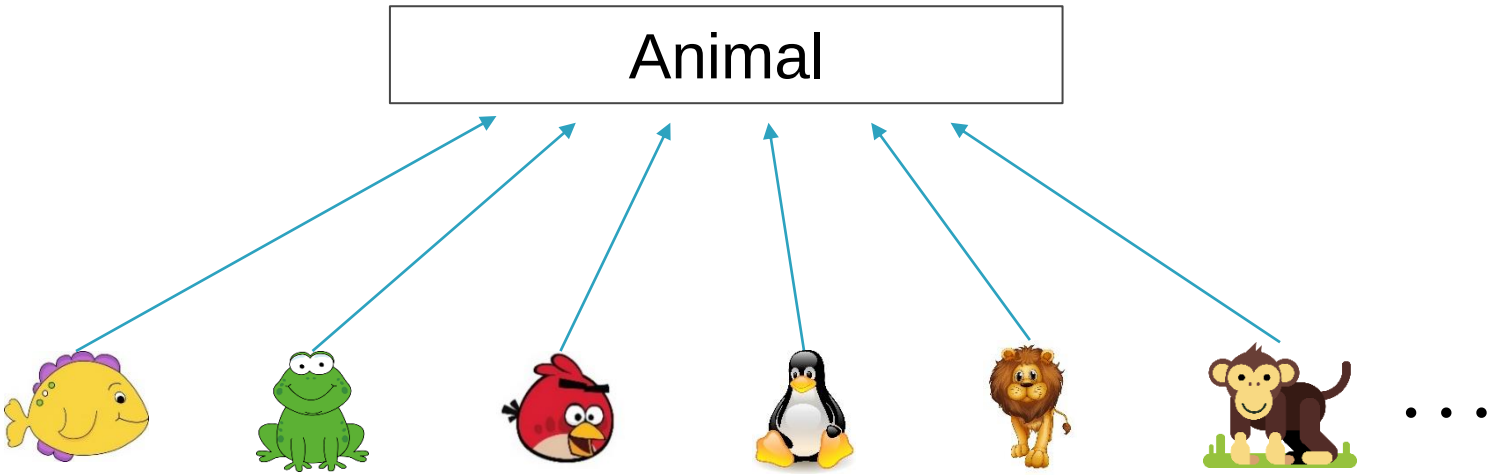
```
manage(new Monkey());
```



Think

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

Concrete Classes & Concrete Methods

- **Concrete class**
  - Classes that provide implementations of every method they declare (some of the implementations can be inherited)
  - Concrete classes can be used to instantiate objects (i.e., we can `new ConcreteClass(…)`)

- **Concrete Methods**
  - Methods with implementations (method body)

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**"
- Abstract clases 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
- 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 Method

- Abstract methods are declared with the keyword `abstract` and provides no implementations.
- Abstract methods specify the common interfaces (method signature) that subclasses need to implement

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

Be careful, no brackets {}

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.
- For the same reason, abstract methods cannot be `static` or `final`

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(); // assume Animal is abstract
```

- When called, such a method can receive an object of any concrete class that directly or indirectly extends the abstract superclass `Animal`.

```
moveAnimal(Animal a) { a.move() }
```

- 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

```
public abstract class Animal {
    int position_x;
    int position_y;

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

    public abstract void move();

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

- Constructors cannot be declared `abstract` (constructors are not inherited)
- The constructor of the abstract class ensures that certain fields are always initialized in a certain way in the concrete subclasses

Using Abstract Classes

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

```
public abstract class Animal {
    int position_x;
    int position_y;

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

    public abstract void move();

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

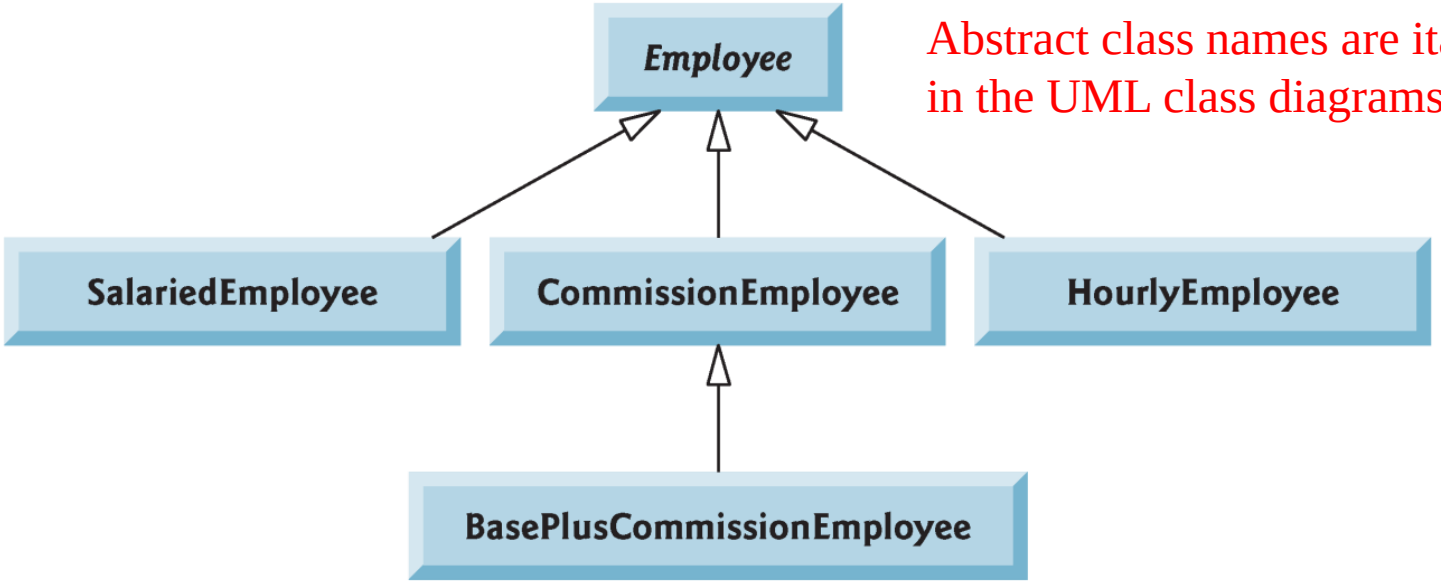
```
public class Fish extends Animal {
    @Override
    public void move(){
        System.out.println("Fish swims");
    }
}
```

```
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

- The company pays its four types of employees on a weekly basis.
  - **Commission employees** are paid a percentage of their sales
  - **Base-plus commission employees** get a base salary + a percentage of their sales.
  - **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
- The company wants to write a Java application that performs its payroll calculations polymorphically.

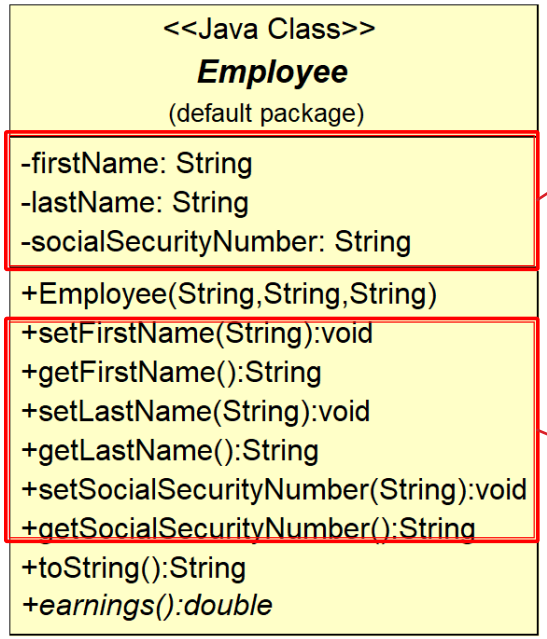
Design: Primary Classes



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.

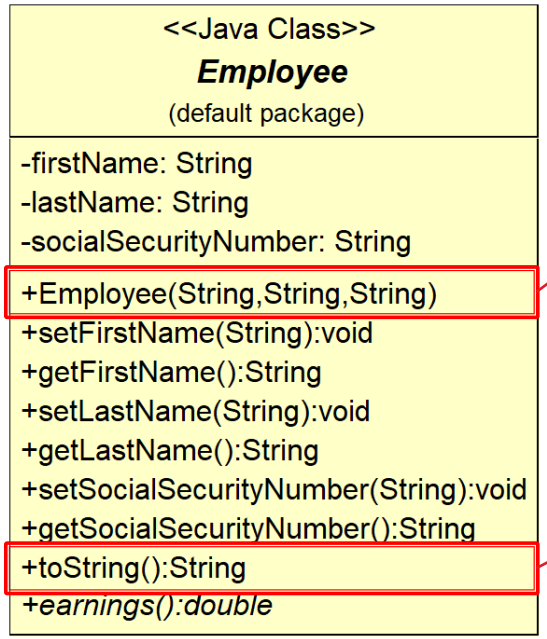


Each employee has a first name, a last name and a social security number. This applies to all employee types.

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.

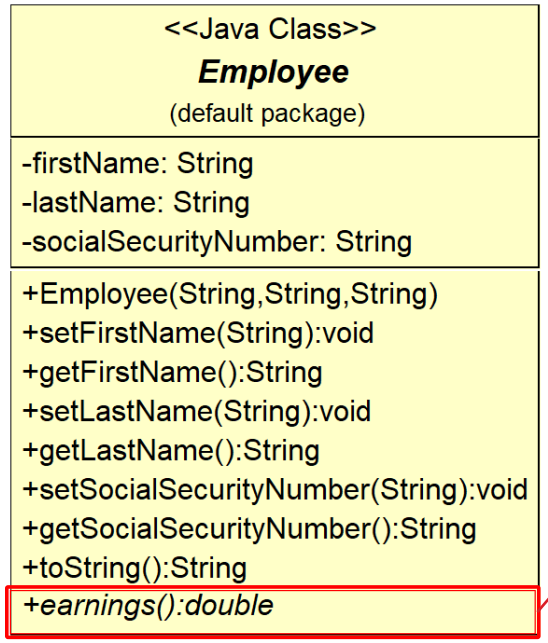


A constructor for initializing the three fields

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.

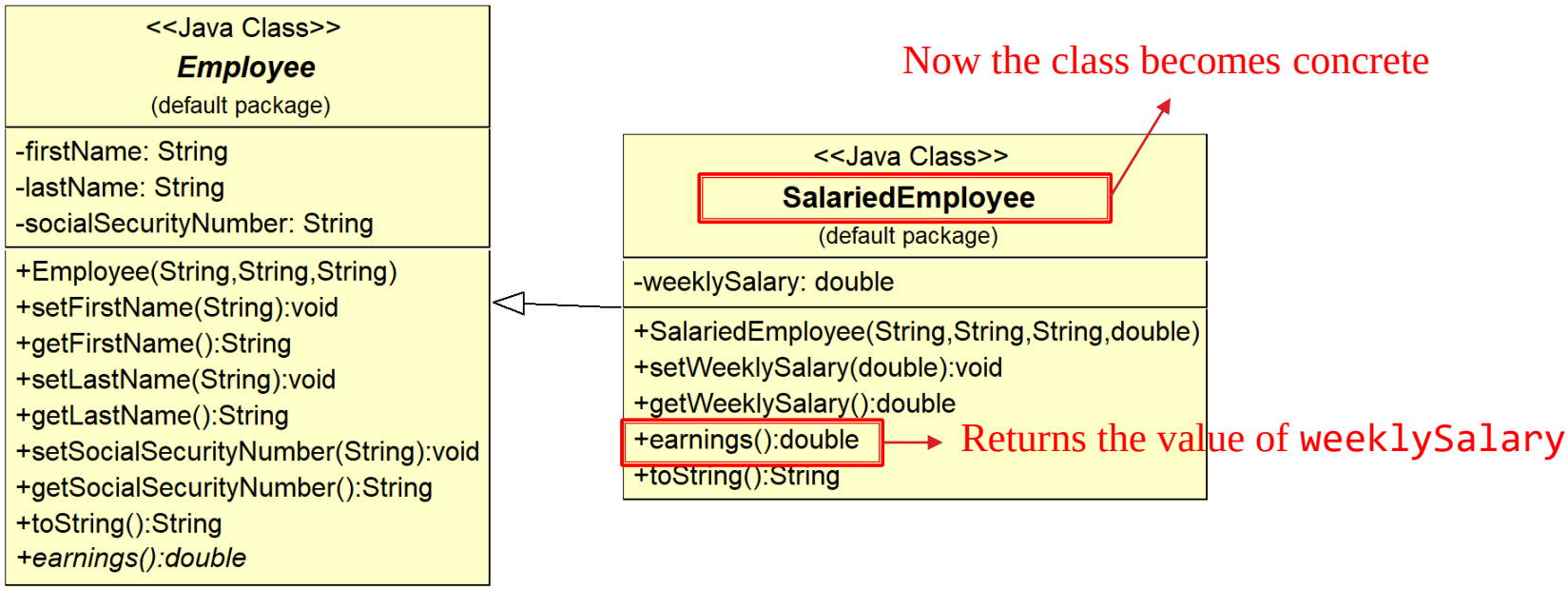


Abstract method names are italicized

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

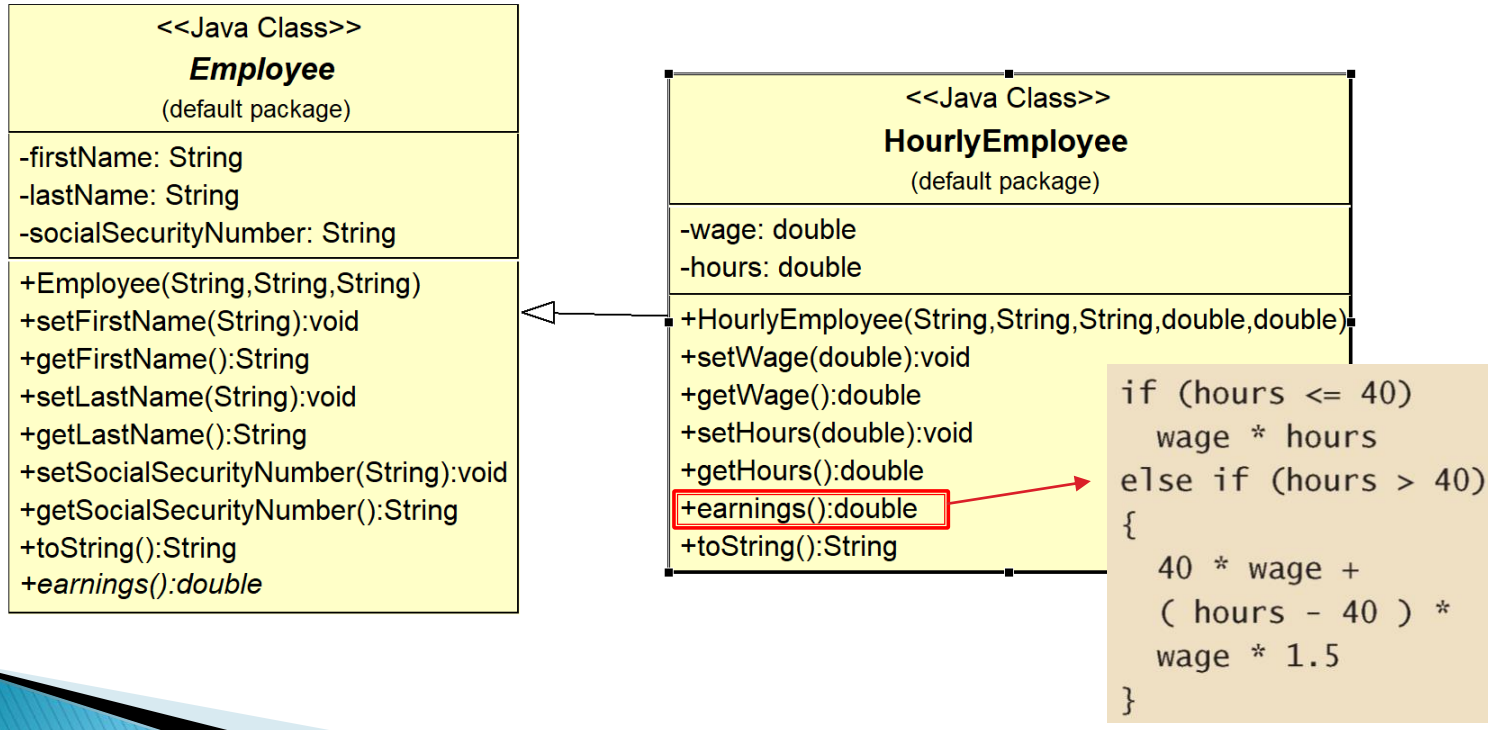
The SalariedEmployee Class

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



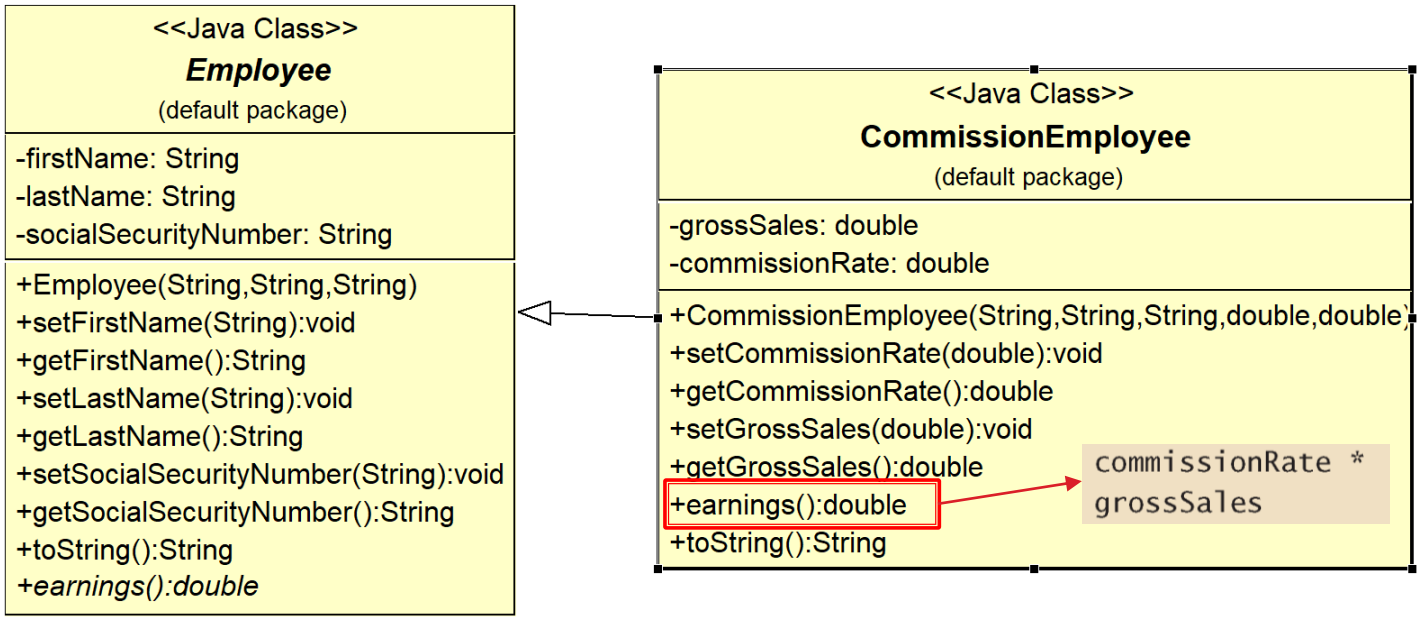
The HourlyEmployee Class

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



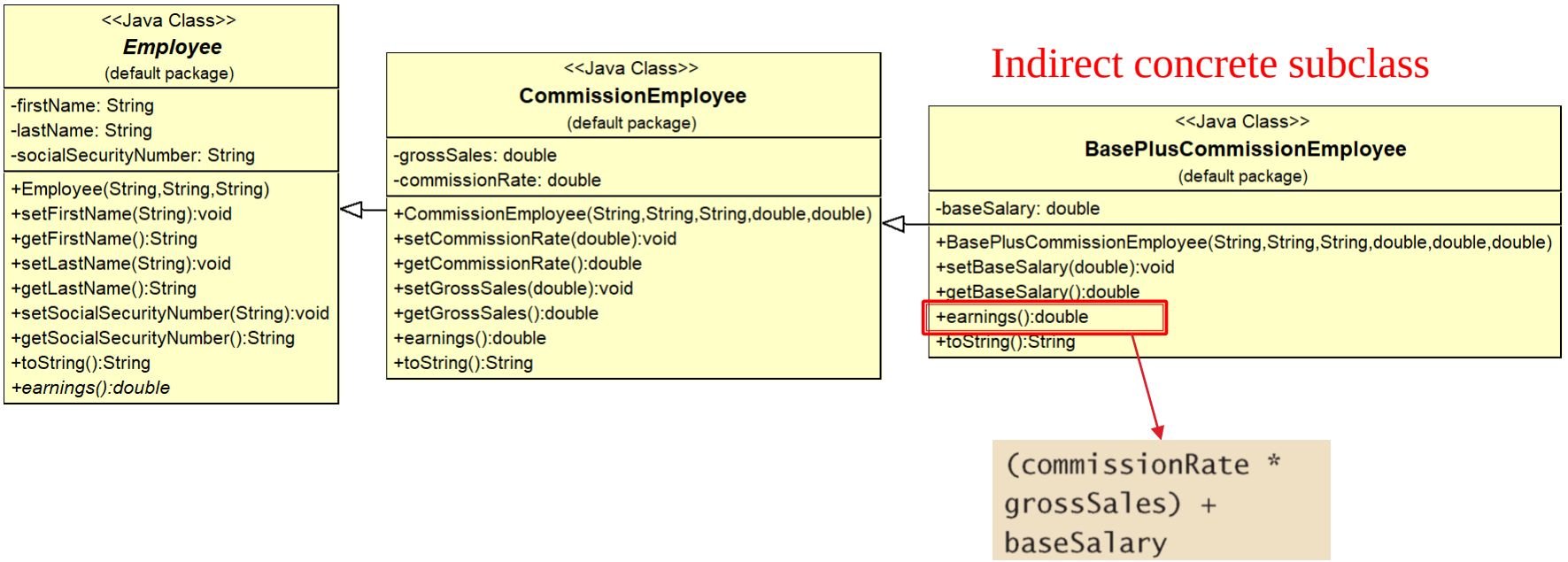
The CommissionEmployee Class

- 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: Design I

```
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**.

Putting Things Together: Design I

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
```


Putting Things Together: Design II

```
// 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() );
}
```

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

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

Putting Things Together: Design II

```
// 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?

Inside the for loop……

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: `currentEmployee` and `employee` refers to the same object

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
```

