جامعة الأميرة نورة بنت عبد الرحمن
Princess Nora Bint Abdul Rahman University

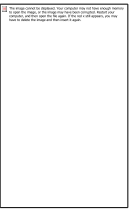
CS111: PROGRAMMING LANGUAGE II

Computer Science Department

Lecture 8(b): Abstract classes & Polymorphism

Lecture Contents

- Abstract base classes
- Concrete classes
- Polymorphic processing



Case Study: Payroll System

A company pays its employees on a weekly basis. The employees are of four types: *Salaried employees* are paid a fixed weekly salary regardless of the number of hours worked, *hourly employees* are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, *commission employees* are paid a percentage of their sales and base-salaried commission employees receive a base salary plus 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.**

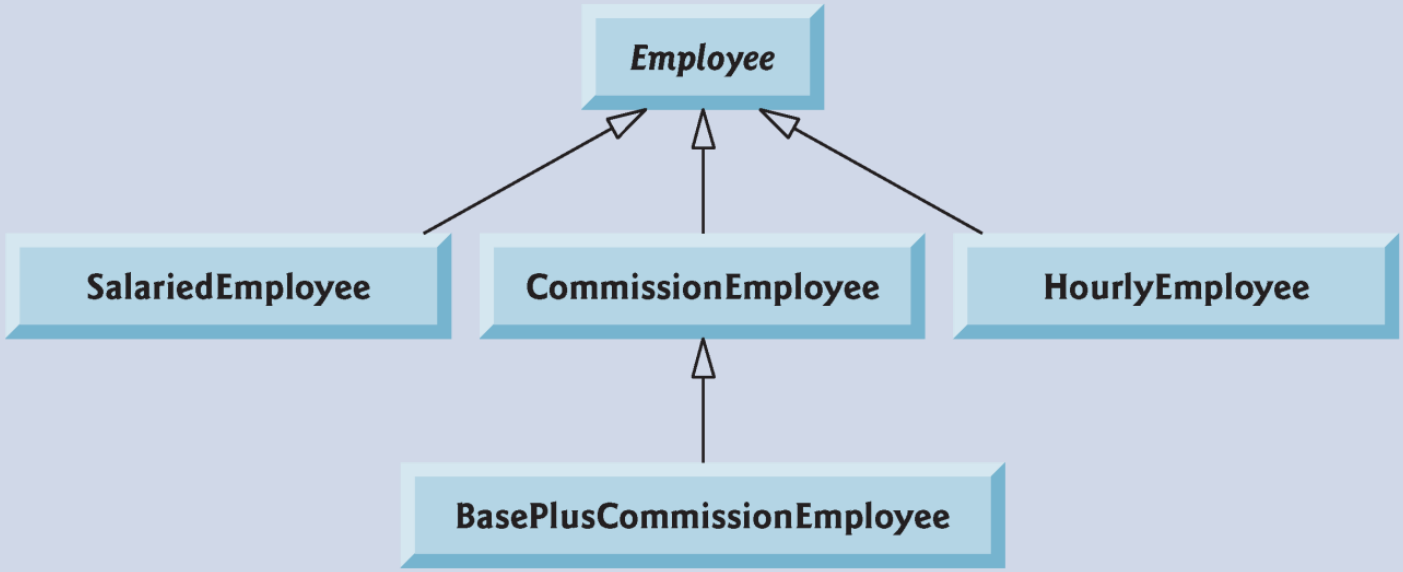


Fig. 10.2 | `Employee` hierarchy UML class diagram.

4

Case Study: Payroll System

Concrete Subclasses

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*

Hourly-Employee	if (hours <= 40)
	  wage * hours
	else if (hours > 40)
	{
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

Polymorphic Processing

Array of references to the base class

Each reference is instantiated as an object of a concrete class.

Dynamic binding → different method calls according to actual type!!

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest
5  {
6     public static void main( String[] args )
7     {
8        // create subclass objects
9        SalariedEmployee salariedEmployee =
10          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11       HourlyEmployee hourlyEmployee =
12          new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13       CommissionEmployee commissionEmployee =
14          new CommissionEmployee(
15          "Sue", "Jones", "333-33-3333", 10000, .06 );
16       BasePlusCommissionEmployee basePlusCommissionEmployee =
17          new BasePlusCommissionEmployee(
18          "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20       System.out.println( "Employees processed individually:\n" );
21
22       System.out.printf( "%s\n%s: $%,.2f\n\n",
23          salariedEmployee, "earned", salariedEmployee.earnings() );
```

Fig. 10.9 | Employee hierarchy test program. (Part 1 of 6.)

```
24          System.out.printf( "%s\n%s: $%,.2f\n\n",
25             hourlyEmployee, "earned", hourlyEmployee.earnings() );
26          System.out.printf( "%s\n%s: $%,.2f\n\n",
27             commissionEmployee, "earned", commissionEmployee.earnings() );
28          System.out.printf( "%s\n%s: $%,.2f\n\n",
29             basePlusCommissionEmployee,
30             "earned", basePlusCommissionEmployee.earnings() );
31
32          // create four-element Employee array
33          Employee[] employees = new Employee[ 4 ];
34
35          // initialize array with Employees
36          employees[ 0 ] = salariedEmployee;
37          employees[ 1 ] = hourlyEmployee;
38          employees[ 2 ] = commissionEmployee;
39          employees[ 3 ] = basePlusCommissionEmployee;
40
41          System.out.println( "Employees processed polymorphically:\n" );
42
43          // generically process each element in array employees
44          for ( Employee currentEmployee : employees )
45          {
46             System.out.println( currentEmployee ); // invokes toString
47
```

Does not create `Employee` objects—just variables that can refer to objects of `Employee` subclasses

Aim each `Employee` variable at an object of an `Employee` subclass

Polymorphically invokes `toString`

Fig. 10.9 | `Employee` hierarchy test program. (Part 2 of 6.)

```
48            // determine whether element is a BasePlusCommissionEmployee
49            if ( currentEmployee instanceof BasePlusCommissionEmployee )
50            {
51               // downcast Employee reference to
52               // BasePlusCommissionEmployee reference
53               BasePlusCommissionEmployee employee =
54                  ( BasePlusCommissionEmployee ) currentEmployee;
55
56               employee.setBaseSalary( 1.10 * employee.getBaseSalary() );
57
58               System.out.printf(
59                  "new base salary with 10%% increase is: $%,.2f\n",
60                  employee.getBaseSalary() );
61            } // end if
62
63            System.out.printf(
64               "earned $%,.2f\n\n", currentEmployee.earnings() );
65         } // end for
66
67         // get type name of each object in employees array
68         for ( int j = 0; j < employees.length; j++ )
69            System.out.printf( "Employee %d is a %s\n", j,
70               employees[ j ].getClass().getName() );
71      } // end main
72   } // end class PayrollSystemTest
```

Is `currentEmployee` a `BasePlus-CommissionEmployee`?

This downcast works because `currentEmployee` is a `BasePlus-CommissionEmployee`

Polymorphically invokes `earnings`

Every object in Java knows its own type

Fig. 10.9 | `Employee` hierarchy test program. (Part 3 of 6.)

```
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
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 4 of 6.)

```
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
```

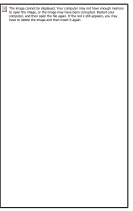
Fig. 10.9 | `Employee` hierarchy test program. (Part 5 of 6.)

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

Fig. 10.9 | `Employee` hierarchy test program. (Part 6 of 6.)

Demonstrating Polymorphic Behavior

- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.



Common Programming Error 10.3

Assigning a superclass variable to a subclass variable (without an explicit cast) is a compilation error.

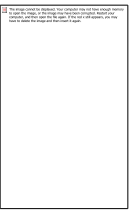
Common Programming Error 10.4

When downcasting an object, a `ClassCastException` *occurs if at execution time the object does not have an* is-a *relationship with the type specified in the cast operator. A reference can be cast only to its own type or to the type of one of its superclasses.*

Programming Pitfalls

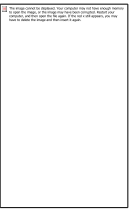
Class Information

- Every object in Java 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 `Class` (from package `java.lang`), which contains information about the object's type, including its class name.
  - The result of the `getclass` call is used to invoke `getName` to get the object's class name.



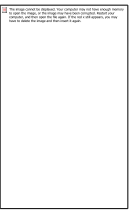
Abstract Classes and Methods

- Programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of subclass types.
  - You can write a method with a parameter of an abstract superclass type.
  - When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.



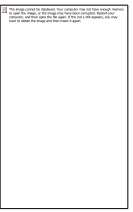
Polymorphism Examples (1)

- 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 the three types of animals under investigation.
  - Each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x*-*y* coordinates. Each subclass implements method `move`.
  - A program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per second—namely, `move`.



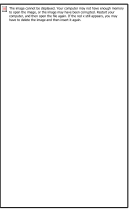
Polymorphism Examples (1)

- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same message (i.e., `move`) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- 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.



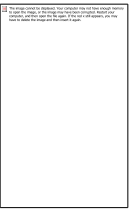
Polymorphism Examples (2)

- Example: Quadrilaterals
  - If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
  - Any operation that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`.
  - These operations can also be performed on other `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.
  - 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 type of the reference stored in the superclass variable.



Benefits of Polymorphism

- With polymorphism, we can design and implement systems that are easily *extensible*
  - 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 that we add to the hierarchy.

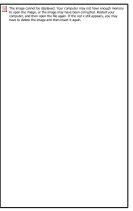


`final` Methods and Classes

- A `final` method in a superclass cannot be overridden in a subclass.
  - Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass.
  - Methods that are declared `static` are implicitly `final`.
  - A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as static binding.

`final` Methods and Classes (Cont.)

- A `final` class cannot be a superclass (i.e., a class cannot extend a `final` class).
  - All methods in a `final` class are implicitly `final`.
- Class `String` is an example of a `final` class.
  - If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected.
  - Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions.



Common Programming Error 10.5

Attempting to declare a subclass of a `final` *class is a compilation error.*

Software Engineering Observation 10.6

In the Java API, the vast majority of classes are not *declared* `final`*. This enables inheritance and polymorphism. However, in some cases, it's important to declare classes* `final`*—typically for security reasons.*

Programming Pitfalls

That's all for today…..

Working on Chapter 10…..