

## CHAPTER7 Inheritance

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of Computer Science and Information Engineering
National Cheng Kung University





- ☐ The sharing of attributes and operations among classes based on a hierarchical relationship
  - ➤ It allows code to be *reused*, without having to copy it into the definitions of the derived classes
- ☐ Each subclass inherits all of the properties of its superclass and adds its own unique properties (called extension)
- ☐ Is-a relationship



## **Introduction to Inheritance**

- ☐ The original class is called the *base class*
- ☐ The new class is called a *derived class* 
  - A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well



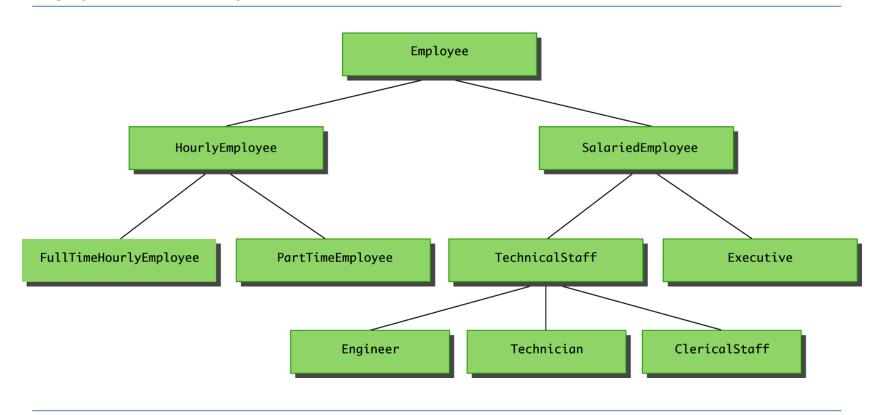
## **Examples of Derived Classes**

- ☐ Within Java, a class called **Employee** can be defined that includes all employees
- ☐ This class can then be used to define classes for hourly employees and salaried employees
  - In turn, the **HourlyEmployee** class can be used to define a **PartTimeHourlyEmployee** class, and so forth

public class HourlyEmployee extends Employee



#### Display 7.1 A Class Hierarchy





## **Derived Class (Subclass)**

- ☐ Members of a class that are declared **private** are not inherited by subclasses of that class.
- ☐ Only members of a class that are declared **protected** or **public** are inherited by subclasses declared in a package other than the one in which the class is declared.

```
import java.util.Date;
public class Employee {
 protected String name;
 protected Date hireDate;
 public Employee(){}
 public Employee(String theName, Date theDate){
       name = theName;
       hireDate = theDate;
 public Date getHireDate(){
        return hireDate;
 public String getName(){
        return name;
```

```
import java.util.Date;

public class HourlyEmployee extends Employee{
   private double wageRate;

public HourlyEmployee(String theName, Date theDate, double rate){
        name = theName;
        hireDate = theDate;
        wageRate = rate;
}
```

```
import java.util.Date;
public class Company {
   public static void main(String[] args){
      HourlyEmployee hourlyEmployee = new HourlyEmployee("Josephine", new Date(114,0,1), 100);
      System.out.println(hourlyEmployee.getName());
   }
}
```



## Parent and Child Classes

- ☐ A base class is often called the *parent class* 
  - A derived class is then called a *child class*
- ☐ These relationships are often extended such that a class that is a parent of a parent . . . of another class is called an *ancestor class* 
  - ➤ If class A is an ancestor of class B, then class B can be called a *descendent* of class A



## Overriding a Method Definition

- ☐ Although a derived class inherits methods from the base class, it can **change** or *override* an inherited method if necessary
  - ➤ In order to override a method definition, a new definition of the method is simply placed in the class definition



```
import java.util.Date;

public class HourlyEmployee extends Employee{
  private double wageRate;

public HourlyEmployee(String theName, Date theDate, double rate){
     name = theName;
     hireDate = theDate;
     wageRate = rate;
}

public String getName(){
    return "Hourly Employee:" + name;
}
}
```

Then run Company again!



## **Changing the Access Permission of an Overridden Method**

- ☐ The access permission of an overridden method can be changed **from private** in the base class **to public** (or some other more permissive access) in the derived class
- However, the access permission of an overridden method can not be changed from public in the base class to a more restricted access permission in the derived class



## **Changing the Access Permission of an Overridden Method**

```
☐ Given the following method header in a base case:
   private void doSomething()
☐ The following method header is valid in a derived class:
   public void doSomething()
☐ Given the following method header in a base case:
   public void doSomething()
☐ The following method header is not valid in a derived
  class:
   private void doSomething()
```



## Pitfall: Overriding Versus Overloading

- ☐ When a method is **overridden**, the new method definition given in the derived class has the **exact same number and types of parameters** as in the base class
- ☐ When a method in a derived class has a **different signature** from the method in the base class, that is **overloading**



### The final Modifier

- ☐ If the modifier **final** is placed before the definition of a *method*, then that method may not be redefined in a derived class
- ☐ It the modifier **final** is placed before the definition of a *class*, then that class may not be used as a base class to derive other classes



```
import java.util.Date;
public class Employee {
 protected String name;
 protected Date hireDate;
 public Employee(){}
 public Employee(String theName, Date theDate){
       name = theName;
       hireDate = theDate;
 public Date getHireDate(){
        return hireDate;
 final public String getName(){
        return name;
```

Then see what happens in HourlyEmployee!



## The super Constructor

- ☐ A derived class uses a constructor from the base class to initialize all the data inherited from the base class
  - ➤ In order to invoke a constructor from the base class, it uses a special syntax:

```
public derivedClass(int p1, int p2, double p3)
{
   super(p1, p2);
   instanceVariable = p3;
}
```

In the above example, super(p1, p2); is a call to the base class constructor



#### Step 1: remove the "final" added in the previous lab.

```
import java.util.Date;

public class HourlyEmployee extends Employee{
    private double wageRate;

public HourlyEmployee(String theName, Date theDate, double rate){
        super(theName,theDate); Step 2: revise code here

        wageRate = rate;
}

public String getName(){
        return "Hourly Employee:" + name;
}
```



## The super Constructor

- ☐ If a derived class constructor does not include an invocation of **super**, then the no-argument constructor of the base class will automatically be invoked
- ☐ Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to super should always be used

```
class A
{
    int a;
    public A()
    {
        a = 7;
        System.out.println("Step 1");
    }
}
```

```
class B extends A
{
   public B()
   {
      System.out.println("Step 2");
   }
   public static void main(String[] args){
      B bo = new B();
   }
}
```



### The this Constructor

- ☐ Often, a no-argument constructor uses this to invoke an explicit-value constructor
  - ➤ No-argument constructor (invokes explicit-value constructor using this and default arguments):

```
public ClassName()
{
   this(argument1, argument2);
}
```

> Explicit-value constructor (receives default values):

```
public ClassName(type1 param1, type2 param2)
{
    . . .
}
```



- ☐ More generally, an object of a derived class has the type of every one of its ancestor classes
  - Therefore, an object of a derived class can be assigned to a variable of any ancestor type
- ☐ An ancestor type can never be used in place of one of its derived types



```
import java.util.Date;
public class Company {
  public static void main(String[] args){
   HourlyEmployee hourlyEmployee = new HourlyEmployee("Josephine",
            Date(114,0,1), 100);
       new
    System.out.println(hourlyEmployee.getName());
    Employee someEmploy = hourlyEmployee;
   printHireDate(someEmploy);
  public static void printHireDate(Employee someEmploy){
   System.out.println(someEmploy.getHireDate());
```

Modifier	Class	Package	Subclass	World
public	V	V	V	V
protected	V	V	V	X
default (package access)	V	V	X	X
private	V	Х	Х	X



- ☐ The protected modifier provides very weak protection compared to the private modifier
  - ➤ It allows direct access for any programmer who defines a suitable derived class
  - > Therefore, instance variables should normally not be marked protected

```
import java.util.Date;
public class Employee {
 private String name;
 private Date hireDate;
 public Employee(){}
 public Employee(String theName, Date theDate){
       name = theName;
       hireDate = theDate;
 public Date getHireDate(){
        return hireDate;
 public String getName(){
        return name;
```



```
import java.util.Date;

public class HourlyEmployee extends Employee{
   private double wageRate;

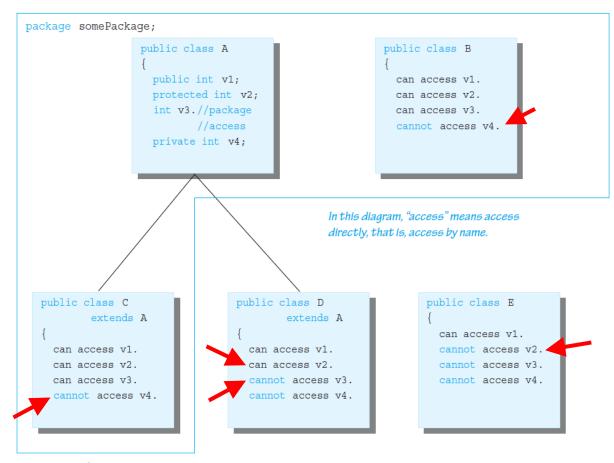
public HourlyEmployee(String theName, Date theDate, double rate){
        super(theName,theDate);
        wageRate = rate;
}

public String getName(){
        return "Hourly Employee:" + super.getName();
}
```



### **Access Modifiers**

Display 7.9 Access Modifiers



A line from one class to another means the lower class is a derived class of the higher class.

If the instance variables are replaced by methods, the same access rules apply.



## Tip: "Is a" Versus "Has a"

- ☐ A derived class demonstrates an "is a" relationship between it and its base class
  - Forming an "is a" relationship is one way to make a more complex class out of a simpler class
  - For example, an HourlyEmployee "is an" Employee
  - ➤ HourlyEmployee is a more complex class compared to the more general Employee class



## Tip: "Is a" Versus "Has a"

- ☐ Another way to make a more complex class out of a simpler class is through a "has a" relationship
  - ➤ This type of relationship, called *composition*, occurs when a class contains an instance variable of a class type
  - The Employee class contains an instance variable, hireDate, of the class Date, so therefore, an Employee 'has a' Date



## Tip: "Is a" Versus "Has a"

- ☐ Both kinds of relationships are commonly used to create complex classes, often within the same class
  - ➤ Since HourlyEmployee is a derived class of Employee, and contains an instance variable of class Date, then HourlyEmployee "is an" Employee and "has a" Date



# You Cannot Use Multiple supers

- ☐ It is only valid to use **super** to invoke a method from a direct parent
  - ➤ Repeating **super** will not invoke a method from some other ancestor class
- ☐ For example, if the Employee class were derived from the class Person, and the HourlyEmployee class were derived form the class Employee, it would not be possible to invoke the toString method of the Person class within a method of the HourlyEmployee class

super.super.toString() // ILLEGAL!



- ☐ In Java, every class is a descendent of the class Object
  - Every class has **Object** as its ancestor
  - Every object of every class is of type Object, as well as being of the type of its own class
- ☐ If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class Object



## The Class Object

- The class **Object** is in the package **java.lang** which is always imported automatically
- Having an **Object** class enables methods to be written with a parameter of type **Object** 
  - A parameter of type Object can be replaced by an object of any class
  - For example, some library methods accept an argument of type Object so they can be used with an argument that is an object of any class



## The Class Object

- ☐ The class Object has some methods that every Java class inherits
  - For example, the **equals** and **toString** methods
- ☐ Every object inherits these methods from some ancestor class
  - Either the class **Object** itself, or a class that itself inherited these methods (ultimately) from the class **Object**
- ☐ However, these inherited methods should be overridden with definitions more appropriate to a given class
  - Some Java library classes assume that every class has its own version of such methods

Modifier and Type	Method and Description
protected Object	<u>clone</u> ()Creates and returns a copy of this object.
boolean	equals(Object obj)Indicates whether some other object is "equal to" this one.
protected void	<u>finalize</u> ()Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class	getClass()Returns the runtime class of this Object.
int	hashCode()Returns a hash code value for the object.
void	notify()Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll()Wakes up all threads that are waiting on this object's monitor.
String	toString()Returns a string representation of the object.
void	wait()Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait (long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	<u>wait</u> (long timeout, int nanos)Causes the current thread to wait until another thread invokes the <u>notify()</u> method or the <u>notifyAll()</u> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.



☐ Since the equals method is always inherited from the class Object, methods like the following simply overload it:

public boolean equals(Employee otherEmployee)
{ . . . }

# Lab

```
import java.util.Date;
public class Employee {
private String name;
private Date hireDate;
public Employee(){}
    public Employee(String theName, Date theDate){
        name = theName;
        hireDate = theDate;
    public Date getHireDate(){
        return hireDate;
    public String getName(){
        return name;
    public boolean equals(Employee otherone){
      if(otherone.getName().equals(this.name) && otherone.getHireDate().equals(this.hireDate)){
        return true;
      }else{
        return false;
```

```
import java.util.Date;

public class CompareTest {

   public static void main(String[] args) {
       Employee employeeA = new Employee("Josephine", new Date(114,0,1));
       Employee employeeB = new Employee("Josephine", new Date(114,0,1));

      System.out.println(employeeA.equals(employeeB));

      System.out.println(employee == employeeB);
   }
}
```



### instanceof and getClass

- □Both the instanceof operator and the getClass() method can be used to check the class of an object
- ☐ However, the **getClass()** method is more exact
  - The instanceof operator simply tests the class of an object
  - The getClass() method used in a test with == or != tests if two objects were created with the same class



# The instanceof Operator

- ☐ The instanceof operator checks if an object is of the type given as its second argument

  Object instanceof ClassName
  - This will return **true** if **Object** is of type **ClassName**, and otherwise return **false**
  - Note that this means it will return **true** if **Object** is the type of any descendent class of **ClassName**



## The getClass() Method

- □ Every object inherits the same getClass() method from the Object class
  - This method is marked **final**, so it cannot be overridden
- An invocation of **getClass()** on an object returns a representation *only* of the class that was used with **new** to create the object
  - The results of any two such invocations can be compared with == or != to determine whether or not they represent the exact same class

```
(object1.getClass() == object2.getClass())
```

```
import java.util.Date;
public class CompareTest {
  public static void main(String[] args) {
    Employee employeeA = new Employee("Josephine", new Date(114,0,1));
    Employee employeeB = new Employee("Josephine", new Date(114,0,1));
    System.out.println(employeeA.equals(employeeB));
    System.out.println(employeeA == employeeB);
    if(employeeA instanceof Employee){
      System.out.println(employeeA.getName() + "is an object of Employee");
    System.out.println(employeeA.getClass().getName());
```



- Inheritance is the process by which a new class known as a \_\_\_\_\_ is created from another class, called the \_\_\_\_\_.

  (a)base class, derived class
  - (b)derived class, base class
  - (c)inherited class, base class
  - (d)base class, inherited class



- Inheritance promotes code \_\_\_\_\_
  - (a)reinvention
  - (b)reuse
  - (c)repeats
  - (d)all of the above



#### The keyword **extends** indicates:

- (a)encapsulation
- (b)polymorphism
- (c)inheritance
- (d)none of the above



A derived class is also called a

- (a)sub class
- (b)super class
- (c)base class
- (d)all of the above



A super class is also called a

- (a)derived class
- (b)dominant class
- (c)sub class
- (d)base class



What does a derived class automatically inherit from the base class?

- (a)instance variables
- (b)static variables
- (c)public methods
- (d)all of the above



If the final modifier is added to the definition of a method, this means:

- (a) The method may be redefined in the derived class.
- (b) The method may be redefined in the sub class.
- (c) The method may not be redefined in the derived class.
- (d) None of the above.



The special syntax for invoking a constructor of the base class is:

- (a)super()
- (b)base()
- (c)parent()
- (d)child()



If an instance variable is not modified by public, protected or private then it is said to have:

- (a)Package access
- (b)Default access
- (c)Friendly access
- (d)All of the above



The class \_\_\_\_\_ is an ancestor class of all Java classes.

- (a)String
- (b)Object
- (c)Math
- (d)JFrame



Consider the classes below, declared in the same file:

Which of the statements below is *false*?

- a. Both variables a and b are instance variables.
- b. After the constructor for class B executes, the variable a will have the value 7.
- c. After the constructor for class B executes, the variable b will have the value 8.
- d. A reference of type A can be treated as a reference of type B.

```
public class Bicycle {
    public int speed;
    public Bicycle(int startSpeed) {
        speed = startSpeed;
    public void applyBrake(int decrement) {
        speed -= decrement;
    public void speedUp(int increment) {
        speed += increment;
```

```
public class MountainBike extends Bicycle {
    public int seatHeight;
    public MountainBike(int startHeight,
                        int startSpeed) {
        super(startSpeed);
        seatHeight = startHeight;
    }
    public void setHeight(int newValue) {
        seatHeight = newValue;
```

```
public class BikeTest {
  public static void main(String[] args) {
    MountainBike myBike = new MountainBike(10,20);
    for(int i=0;i<100;i++){</pre>
      myBike.speedUp(1);
    System.out.println(myBike.speed);
    myBike.setHeight(10);
    System.out.println(myBike.seatHeight);
```



- ☐ "Absolute Java". Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- ☐ "Java How to Program". Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- □ "A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition". Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008