# Chapter 7
# Inheritance

Prof. Yongsu Park
Dept. of Computer Science and Engineering
Hanyang University

# Contents

- Overview of inheritance
- Sub-topics w.r.t. inheritances
- The protected access modifiers and package access
- The Class Object

# Overview of inheritance

- Introduction to inheritance
- Derived classes
- Subclasses
- parent/child classes

# Introduction to Inheritance

- *Inheritance* is one of the main techniques of object-oriented programming (OOP)
- Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
  - The specialized classes are said to *inherit* the methods and instance variables of the general class

# Introduction to Inheritance

- Inheritance is the process by which a new class is created from another class
  - The new class is called a *derived class*
  - The original class is called the *base 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
- Inheritance is especially advantageous because it allows code to be *reused*, without having to copy it into the definitions of the derived classes

# Derived Classes

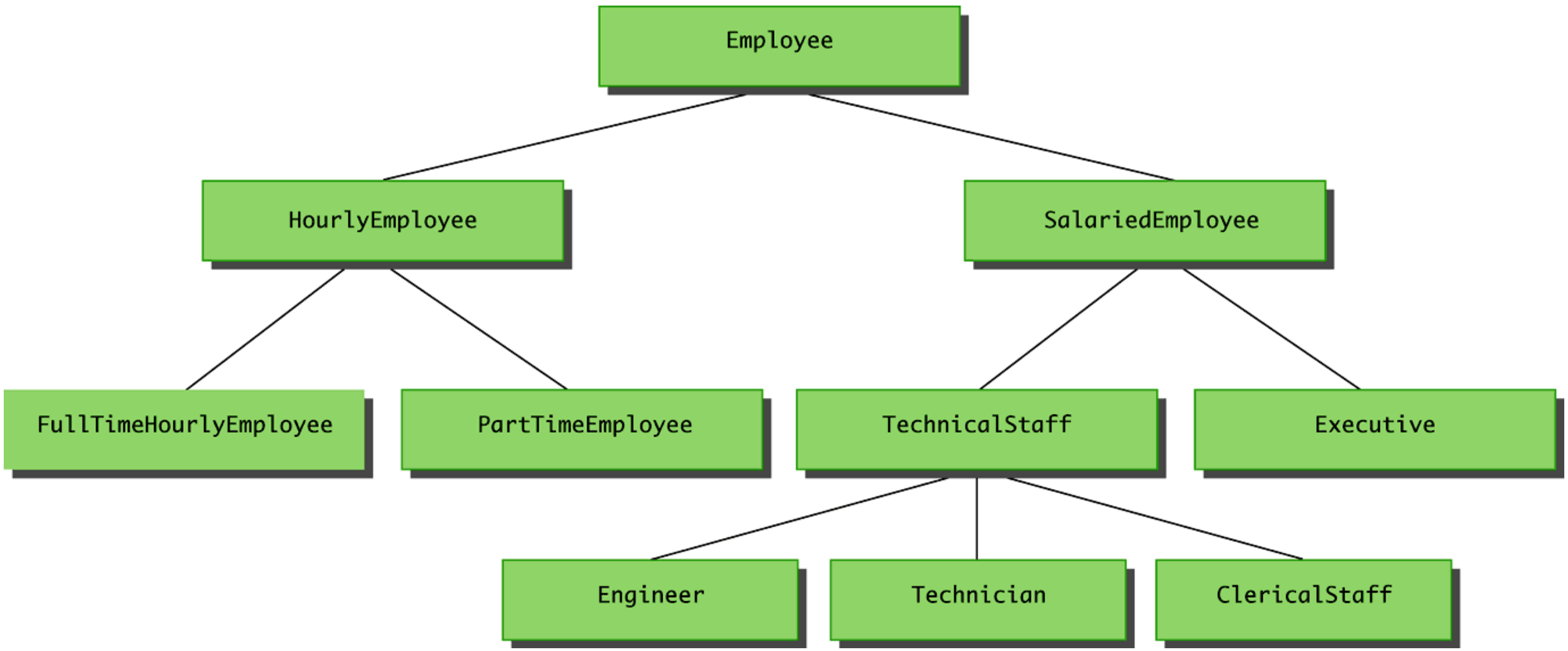
- When designing certain classes, there is often a natural hierarchy for grouping them
  - In a record-keeping program for the employees of a company, there are hourly employees and salaried employees
  - Hourly employees can be divided into full time and part time workers
  - Salaried employees can be divided into those on technical staff, and those on the executive staff

# Derived Classes

- All employees share certain characteristics in common
  - All employees have a name and a hire date
  - The methods for setting and changing names and hire dates would be the same for all employees
- Some employees have specialized characteristics
  - Hourly employees are paid an hourly wage, while salaried employees are paid a fixed wage
  - The methods for calculating wages for these two different groups would be different

# A Class Hierarchy

**Display 7.1 A Class Hierarchy**



# 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

# Derived Classes

- Since an hourly employee is an employee, it is defined as a *derived* class of the class **`Employee`**
  - A *derived class* is defined by adding instance variables and methods to an existing class
  - The existing class that the derived class is built upon is called the *base class*
  - The phrase **`extends BaseClass`** must be added to the derived class definition:

```
public class HourlyEmployee extends Employee
```

# Derived Classes

- When a derived class is defined, it is said to inherit the instance variables and methods of the base class that it extends
  - Class **`Employee`** defines the instance variables **`name`** and **`hireDate`** in its class definition
  - Class **`HourlyEmployee`** also has these instance variables, but they are not specified in its class definition
  - Class **`HourlyEmployee`** has additional instance variables **`wageRate`** and **`hours`** that are specified in its class definition

# Derived Classes

- Just as it inherits the instance variables of the class **`Employee`**, the class **`HourlyEmployee`** inherits all of its methods as well
  - The class **`HourlyEmployee`** inherits the methods **`getName`**, **`getHireDate`**, **`setName`**, and **`setHireDate`** from the class **`Employee`**
  - Any object of the class **`HourlyEmployee`** can invoke one of these methods, just like any other method

# Display 7.2 The Base Class Employee

*The class Date is defined in Display 4.13.*

```
public class Employee
{
    private String name;
    private Date hireDate;

    public Employee()
    {
        name = "No name";
        hireDate = new Date("January", 1, 1000); //Just a placehol
    }

    /**
     Precondition: Neither theName nor theDate is null.
    */
    public Employee(String theName, Date theDate)
    {
        if (theName == null || theDate == null)
        {
            System.out.println("Fatal Error creating employee.");
            System.exit(0);
        }
        name = theName;
        hireDate = new Date(theDate);
    }

    public Employee(Employee originalObject)
    {
        name = originalObject.name;
        hireDate = new Date(originalObject.hireDate);
    }

    public String getName()
    {
        return name;
    }

    public Date getHireDate()
    {
        return new Date(hireDate);
    }
```

```
    /**
     Precondition newName is not null.
    */
    public void setName(String newName)
    {
        if (newName == null)
        {
            System.out.println("Fatal Error sett
            System.exit(0);
        }
        else
            name = newName;
    }

    /**
     Precondition newDate is not null.
    */
    public void setHireDate(Date newDate)
    {
        if (newDate == null)
        {
            System.out.println("Fatal Error sett
            System.exit(0);
        }
        else
            hireDate = new Date(newDate);
    }

    public String toString()
    {
        return (name + " " + hireDate.toString()
    }

    public boolean equals(Employee otherEmployee)
    {
        return (name.equals(otherEmployee.name)
                         && hireDate.equals(otherEm
    }
}
```

# Display 7.3 The Derived Class HourlyEmployee

```java
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee()
    {
        super();
        wageRate = 0;
        hours = 0;
    }

    /**
     Precondition: Neither theName nor theDate is null;
     theWageRate and theHours are nonnegative.
    */
    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        super(theName, theDate);
        if ((theWageRate >= 0) && (theHours >= 0))
        {
            wageRate = theWageRate;
            hours = theHours;
        }
        else
        {
            System.out.println(
                      "Fatal Error: creating an illegal hourly employee.");
            System.exit(0);
        }
    }

    public HourlyEmployee(HourlyEmployee originalObject)
    {
        super(originalObject);
        wageRate = originalObject.wageRate;
        hours = originalObject.hours;
    }
```

*It will take the rest of Section 7.1 to explain this class definition.*

*If this line is omitted, Java will still invoke the no-argument constructor for the base class.*

*An object of the class* **HourlyEmployee** *is also an instance of the class* **Employee.**

# Display 7.3 The Derived Class HourlyEmployee

```java
public double getRate()
{
    return wageRate;
}

public double getHours()
{
    return hours;
}

/**
 Returns the pay for the month.
*/
public double getPay()
{
    return wageRate*hours;
}

/**
 Precondition: hoursWorked is nonnegative.
*/
public void setHours(double hoursWorked)
{
    if (hoursWorked >= 0)
        hours = hoursWorked;
    else
    {
        System.out.println("Fatal Error: Negative hours worked.");
        System.exit(0);
    }
}
```

```java
/**
 Precondition: newWageRate is nonnegative.
*/
public void setRate(double newWageRate)
{
    if (newWageRate >= 0)
        wageRate = newWageRate;
    else
    {
        System.out.println("Fatal Error: Negative wage rate.");
        System.exit(0);
    }
}

public String toString()
{
    return (getName() + " " + getHireDate().toString()
        + "\n$" + wageRate + " per hour for " + hours + " hours");
}

public boolean equals(HourlyEmployee other)
{
    return (getName().equals(other.getName())
        && getHireDate().equals(other.getHireDate())
        && wageRate == other.wageRate
        && hours == other.hours);
}
```

*We will show you a better way to d*
*equals later in this chapter.*

# Derived Class (Subclass)

- A derived class, also called a *subclass*, is defined by starting with another already defined class, called a *base class* or *superclass*, and adding (and/or changing) methods, instance variables, and static variables
  - The derived class inherits all the public methods, all the public and private instance variables, and all the public and private static variables from the base class
  - The derived class can add more instance variables, static variables, and/or methods

# 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`**

# Sub-topics w.r.t. inheritances

- Overriding methods
- The final modifier
- The super constructor
- The this constructor
- An Enhanced StringTokenizer Class
- Access to a Redefined Base Method

# 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, just like any other method that is added to the derived class

# Pitfall: Overriding Versus Overloading

- Do not confuse *overriding* a method in a derived class with *overloading* a method name
  - 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
  - Note that when the derived class overloads the original method, it still inherits the original method from the base class as well

# 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

# 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

# The `super` Constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword `super` instead
- A call to `super` must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to `super`

# 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
  - This can result in an error if the base class has not defined a no-argument constructor
- 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

# The **`this`** Constructor

- Within the definition of a constructor for a class, **`this`** can be used as a name for invoking another constructor in the same class
  - The same restrictions on how to use a call to **`super`** apply to the **`this`** constructor
- If it is necessary to include a call to both **`super`** and **`this`**, the call using **`this`** must be made first, and then the constructor that is called must call **`super`** as its first action

# 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)
{
  . . .
}
```

# The **`this`** Constructor

```
public HourlyEmployee()
{
   this("No name", new Date(), 0, 0);
}
```

- The above constructor will cause the constructor with the following heading to be invoked:

```
public HourlyEmployee(String theName,
  Date theDate, double theWageRate, double
  theHours)
```

# An Enhanced `StringTokenizer` Class

- Thanks to inheritance, most of the standard Java library classes can be enhanced by defining a derived class with additional methods
- For example, the `StringTokenizer` class enables all the tokens in a string to be generated one time
  - However, sometimes it would be nice to be able to cycle through the tokens a second or third time

# Access to a Redefined Base Method

- Within the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked
  - Simply preface the method name with super and a dot

```
public String toString()
{
  return (super.toString() + "$" + wageRate);
}
```

- However, using an object of the derived class outside of its class definition, there is no way to invoke the base class version of an overridden method

# The protected access modifiers and package access

- Private instance variables/method
- Protected and package access
- Access modifiers

# Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from the Base Class

- An instance variable that is private in a base class is not accessible *by name* in the definition of a method in any other class, not even in a method definition of a derived class
  - For example, an object of the **`HourlyEmployee`** class cannot access the private instance variable **`hireDate`** by name, even though it is inherited from the **`Employee`** base class
- Instead, a private instance variable of the base class can only be accessed by the public accessor and mutator methods defined in that class
  - An object of the **`HourlyEmployee`** class can use the **`getHireDate`** or **`setHireDate`** methods to access **`hireDate`**

# Pitfall: Private Methods Are Effectively Not Inherited

- The private methods of the base class are like private variables in terms of not being directly available
- However, a private method is completely unavailable, unless invoked indirectly
  - This is possible only if an object of a derived class invokes a public method of the base class that happens to invoke the private method
- This should not be a problem because private methods should just be used as helping methods
  - If a method is not just a helping method, then it should be public, not private

# `Protected` and `Package` Access

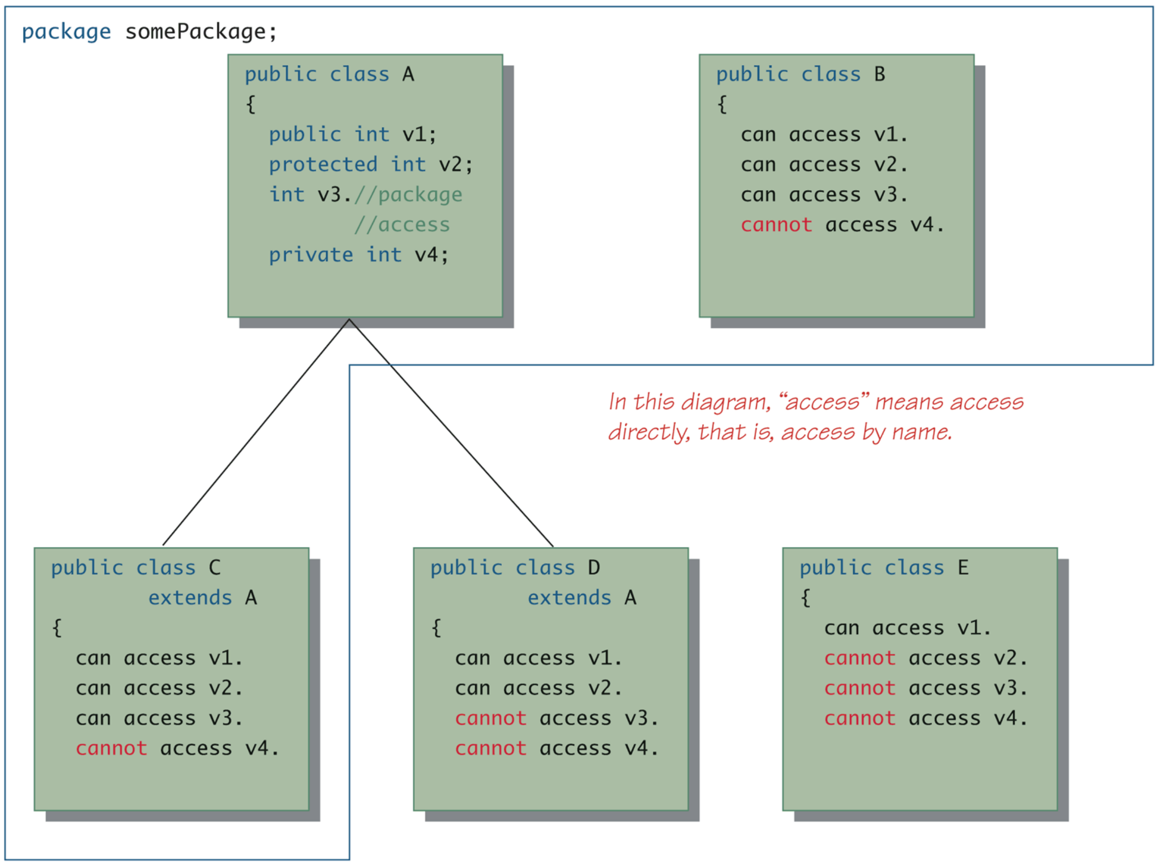
- If a method or instance variable is modified by **`protected`** (rather than **`public`** or **`private`**), then it can be accessed *by name*
  - Inside its own class definition
  - Inside any class derived from it
  - In the definition of any class in the same package
- The **`protected`** modifier provides very weak protection compared to the **`private`** modifier
  - It allows direct access to any programmer who defines a suitable derived class
  - Therefore, instance variables should normally not be marked **`protected`**

# `Protected` and `Package` Access

- An instance variable or method definition that is not preceded with a modifier has *package access*
  - Package access is also known as *default* or *friendly access*
- Instance variables or methods having package access can be accessed *by name* inside the definition of any class in the same package
  - However, neither can be accessed outside the package

# 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.*

# The Class Object

- The Class Object
- The right way to define equals
- The instanceof operator
- The getClass() method

# The Class **`Object`**

- 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 whatsoever
  - 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

# The Right Way to Define `equals`

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

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

- However, this method should be overridden, not just overloaded:

```
public boolean equals(Object otherObject)
{ . . . }
```

# The Right Way to Define `equals`

- The overridden version of `equals` must meet the following conditions
  - The parameter `otherObject` of type `Object` must be type cast to the given class (e.g., `Employee)`
  - However, the new method should only do this if `otherObject` really is an object of that class, and if `otherObject` is not equal to `null`
  - Finally, it should compare each of the instance variables of both objects

# A Better `equals` Method for the Class `Employee`

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;
  else if(getClass( ) != otherObject.getClass( ))
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name) &&
      hireDate.equals(otherEmployee.hireDate));
  }
}
```

# 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())
```

# 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`

# Tip: `getClass` Versus `instanceof`

- Many authors suggest using the `instanceof` operator in the definition of `equals`
  - Instead of the `getClass()` method
- The `instanceof` operator will return `true` if the object being tested is a member of the class for which it is being tested
  - However, it will return `true` *if it is a descendent of that class* as well
- It is possible (and especially disturbing), for the `equals` method to behave inconsistently given this scenario

# Tip: `getClass` Versus `instanceof`

- Here is an example using the class **`Employee`**

```
. . . //excerpt from bad equals method
else if(!(OtherObject instanceof Employee))
  return false; . . .
```

- And an example using the class **`HourleyEmployee`**

```
. . . //excerpt from bad equals method
else if(!(OtherObject instanceof HourleyEmployee))
  return false; . . .
```

- Now consider the following:

```
Employee e = new Employee("Joe", new Date());
HourlyEmployee h = new
  HourlyEmployee("Joe", new Date(),8.5, 40);
boolean testH = e.equals(h);
boolean testE = h.equals(e);
```

# Tip: `getClass` Versus `instanceof`

- `testH` will be `true`, because `h` is an `Employee` with the same name and hire date as `e`
- However, `testE` will be `false`, because `e` is not an `HourlyEmployee`, and cannot be compared to `h`
- Note that this problem would not occur if the `getClass()` method were used instead, as in the previous `equals` method example

# `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