Inheritance

Lecture 9
Object-Oriented Programming

Agenda

- Inheritance
- Motivation
- Code Example
- Object-oriented terminology
- Case Study of a class roaster
- The Protected Modifier
- The instanceof Operator
- Inheritance and Member Accessibility
- The Effect of Three Visibility
- Inheritance and Constructors
- Abstract Superclasses and Abstract Methods
- Inheritance as Form of Abstraction
- Inherit This!
- Is-a Versus Has-a Relationships
- Initializing Data Fields in a Subclass
- Method Overriding
- Method Overloading

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class

Inheritance

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Motivation

- Consider a transportation computer game
  - Different types of vehicles:
    - Planes
      - Jets, helicopters, space shuttle
    - Automobiles
      - Cars, trucks, motorcycles
    - Trains
      - Diesel, electric, monorail
    - Ships
      - …
- Let's assume a class is written for each type of vehicle

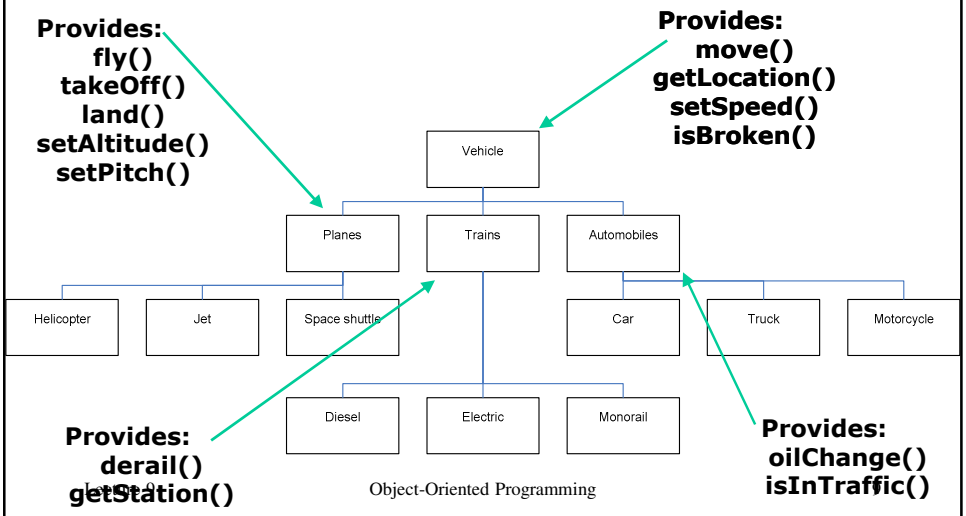
Motivation

- Sample code for the types of planes:
  - fly()
  - takeOff()
  - land()
  - setAltitude()
  - setPitch()
- Note that a lot of this code is common to all types of planes
  - They have a lot in common!
  - It would be a waste to have to write separate fly() methods for each plane type
    - What if you then have to change one – you would then have to change dozens of methods

Motivation

- Indeed, all vehicles will have similar methods:
  - move()
  - getLocation()
  - setSpeed()
  - isBroken()
- Again, a lot of this code is common to all types of vehicles
  - It would be a waste to have to write separate move() methods for each vehicle type
    - What if you then have to change one – you would then have to change dozens of methods
- What we want is a means to specify one move() method, and have each vehicle type ***inherit*** that code
  - Then, if we have to change it, we only have to change one copy

Motivation



Motivation

- What we will do is create a "parent" class and a "child" class
- The "child" class (or subclass) will inherit the methods (etc.) from the "parent" class (or superclass)
- Note that some classes (such as Train) are both subclasses and superclasses

Inheritance code

```
class Vehicle {
    ...
}

class Train extends Vehicles {
  ...
}

class Monorail extends Train {
  ...
}
```

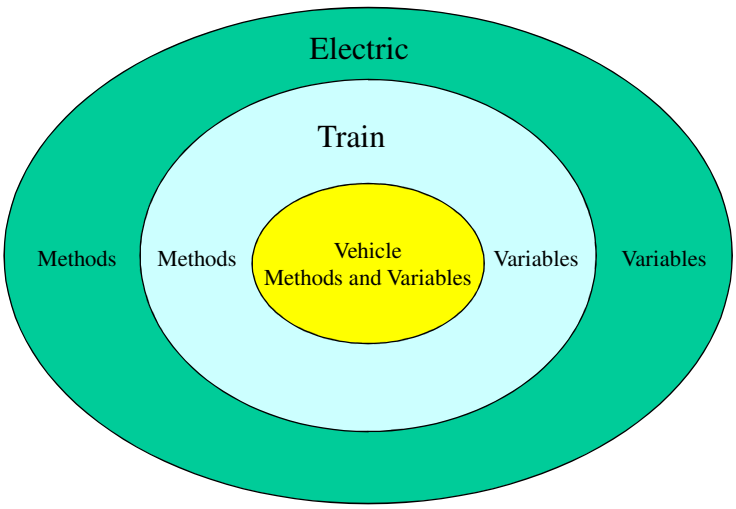

About extends

- If class A extends class B
  - Then class A is the subclass of B
  - Class B is the superclass of class A
  - A "is a" B
  - A has (almost) all the methods and variables that B has
- If class Train extends class Vehicle
  - Then class Train is the subclass of Vehicle
  - Class Vehicle is the superclass of class Train
  - Train "is a" Vehicle
  - Train has (almost) all the methods and variables that Vehicle has

Object-oriented terminology

- In object-oriented programming languages, a class created by extending another class is called a *subclass*
- The class used for the basis is called the *superclass*
- Alternative terminology
  - The superclass is also referred to as the *base* class
  - The subclass is also referred to as the *derived* class

Like an Onion



Defining Classes with Inheritance

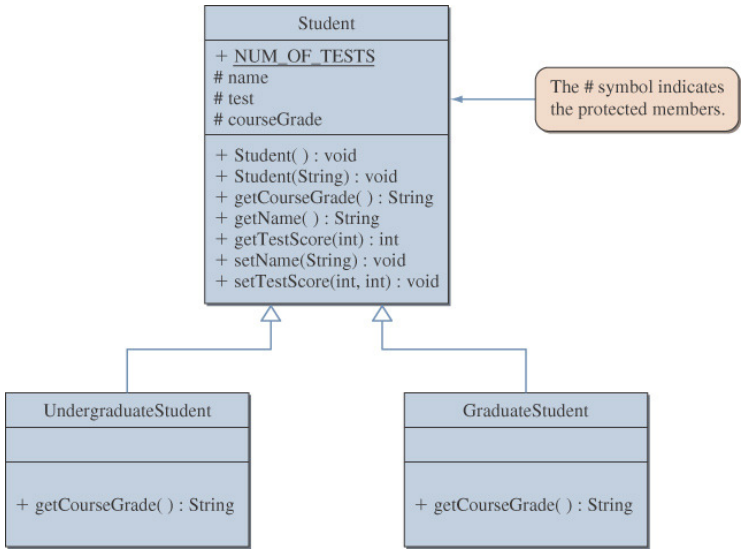
- Case Study:
  - Suppose we want implement a class roster that contains both undergraduate and graduate students.
  - Each student's record will contain his or her name, three test scores, and the final course grade.
  - The formula for determining the course grade is different for graduate students than for undergraduate students.

Modeling Two Types of Students

- There are two ways to design the classes to model undergraduate and graduate students.
  - We can define two unrelated classes, one for undergraduates and one for graduates.
  - We can model the two kinds of students by using classes that are related in an inheritance hierarchy.
- Two classes are *unrelated* if they are not connected in an inheritance relationship.

Classes for the Class Roster

- For the Class Roster sample, we design three classes:
  - Student
  - UndergraduateStudent
  - GraduateStudent
- The **Student** class will incorporate behavior and data common to both **UndergraduateStudent** and **GraduateStudent** objects.
- The **UndergraduateStudent** class and the **GraduateStudent** class will each contain behaviors and data specific to their respective objects.



The `Protected` Modifier

- The modifier **`protected`** makes a data member or method visible and accessible to the instances of the class and the descendant classes.
- **`public`** data members and methods are accessible to everyone.
- **`private`** data members and methods are accessible only to instances of the class.

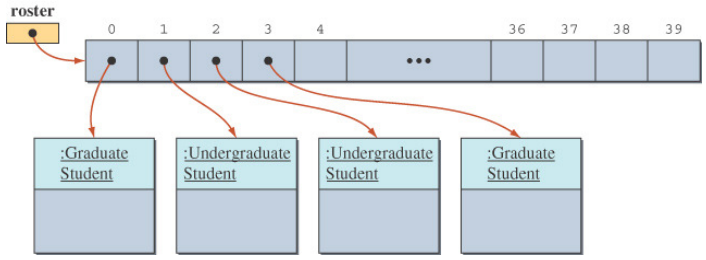
Creating the roster Array

- We can maintain our class roster using an array, combining objects from the **Student**, **UndergraduateStudent**, and **GraduateStudent** classes.

```
Student roster = new Student[40];
. . .
roster[0] = new GraduateStudent();
roster[1] = new UndergraduateStudent();
roster[2] = new UndergraduateStudent();
. . .
```

State of the roster Array

- The roster array with elements referring to instances of **GraduateStudent** or **UndergraduateStudent** classes.



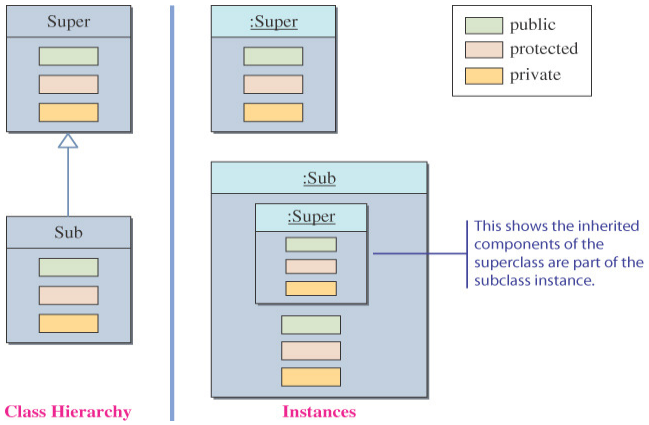
The `instanceof` Operator

- The instanceof operator can help us learn the class of an object.
- The following code counts the number of undergraduate students.

```
int undergradCount = 0;
for (int i = 0; i < numberOfStudents; i++) {
    if ( roster[i] instanceof UndergraduateStudent ) {
        undergradCount++;
    }
}
```

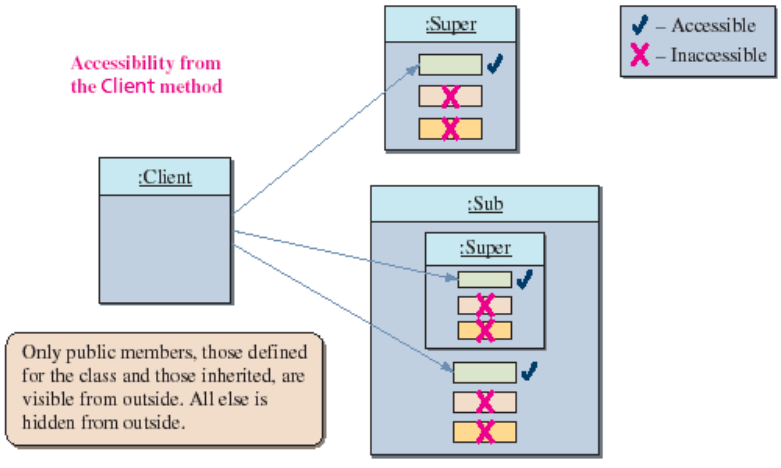
Inheritance and Member Accessibility

- We use the following visual representation of inheritance to illustrate data member accessibility.



23

The Effect of Three Visibility



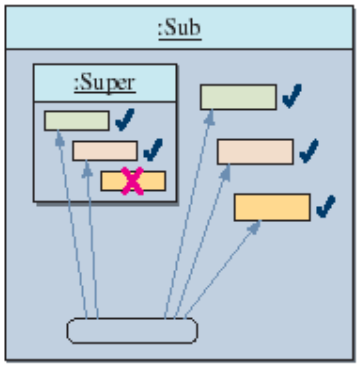
Accessibility of Super from Sub

- Everything except the private members of the Super class is visible from a method of the Sub class.

Accessibility from a method of the Sub class

✓ – Accessible
✗ – Inaccessible

From a method of **Sub**, everything is visible except the private members of its superclass.

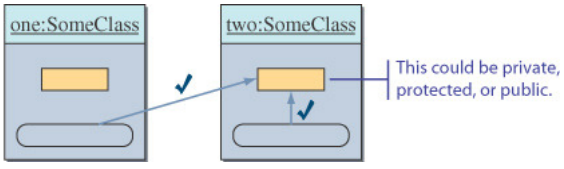


25

Accessibility from Another Instance

- Data members accessible from an instance are also accessible from other instances of the same class.

If a data member is accessible from **anInstance**, that data member is also accessible from **anotherInstance**.



Inheritance Quiz

```
public class A {
  public A() { System.out.println("I'm A"); }
}

public class B extends A {
  public B() { System.out.println("I'm B"); }
}

public class C extends B {
  public C() { System.out.println("I'm C"); }
}
```

What does this print out?

```
C x = new C();
```

```
I'm A
I'm B
I'm C
```


Inheritance and Constructors

- Unlike members of a superclass, constructors of a superclass are *not* inherited by its subclasses.
- You must define a constructor for a class or use the default constructor added by the compiler.
- The statement

  ```
  super();
  ```

  calls the superclass's constructor.
- If the class declaration does not explicitly designate the superclass with the **`extends`** clause, then the class's superclass is the **`Object`** class.

Abstract Superclasses and Abstract Methods

- When we define a superclass, we often do not need to create any instances of the superclass.
- Depending on whether we need to create instances of the superclass, we must define the class differently.
- We will study examples based on the **Student** superclass defined earlier.

Definition: Abstract Class

- An *abstract class* is a class
  - defined with the modifier `abstract` OR
  - that contains an abstract method OR
  - that does not provide an implementation of an inherited abstract method
- An abstract method is a method with the keyword **abstract**, and it ends with a semicolon instead of a method body.
  - Private methods and static methods may not be declared **abstract**.
- No instances can be created from an abstract class.

Case Study - Case 1

- Student Must Be Undergraduate or Graduate
  - If a student must be either an undergraduate or a graduate student, we only need instances of UndergraduateStudent or GraduateStudent.
  - Therefore, we must define the Student class so that no instances may be created of it.

Case Study - Case 2

- Student Does Not Have to Be Undergraduate or Graduate.
- In this case, we may design the Student class in one of two ways.
  - We can make the Student class instantiable.
  - We can leave the Student class abstract and add a third subclass, OtherStudent, to handle a student who does not fall into the UndergraduateStudent or GraduateStudent categories.

Inheritance as Form of Abstraction

- The root of a class hierarchy is the most general object, because it is the superclass to every other object in the hierarchy
  - can always say much more about how a subclass behaves than how its superclass behaves
  - e.g., we can say more about how a `Van` behaves than how a `Car` behaves!

Inheritance: extend classes by adding methods and fields

Inherit This!

5 things you might find in an *Inheritance Hierarchy*:

1) superclass is too general to declare all behaviors, so each subclass adds its own behavior
2) superclass legislates an *abstract* behavior and therefore delegates implementation to subclasses
3) superclass specifies behavior, subclasses inherit behavior
4) superclass specifies behavior, subclasses can choose to *override* behavior
   - **just because a subclass inherits a method doesn't mean that it must act in the same way as its superclass**
   - **subclass can choose to reject its superclass' implementation of any method and "do it my way"**
5) superclass specifies behavior, subclasses can choose to override behavior in part
   - **called *partial overriding***

Is-a Versus *Has-a* Relationships

- Confusing *has-a* and *is-a* leads to misusing inheritance
- Model a *has-a* relationship with an *attribute* (variable)

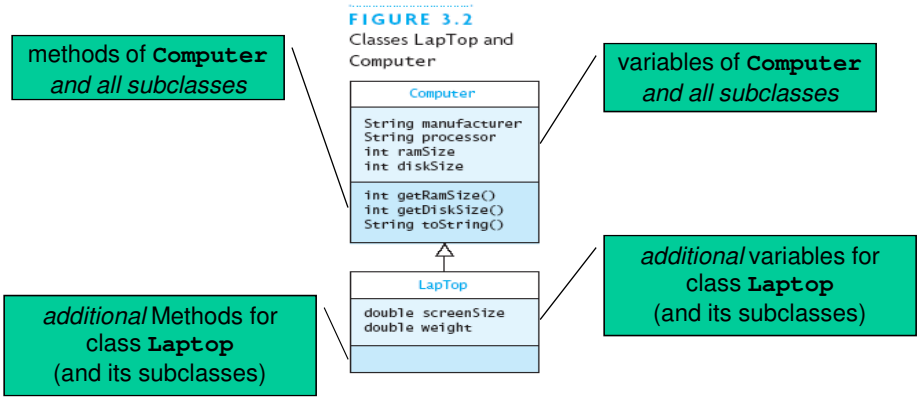
```
public class C { ... private B part; ...}
```

- Model an *is-a* relationship with inheritance
  - If every **C** is-a **B** then model **C** as a subclass of **B**
  - Show this: in **C** include **extends B**:

```
public class C extends B { ... }
```


A Superclass and a Subclass: Code Example

- Consider two classes: **Computer** and **Laptop**
- A laptop is a *kind* of computer: therefore a subclass



Illustrating *Has-a* with `Computer`

```
public class Computer {
  private Memory mem;
  ...
}

public class Memory {
  private int size;
  private int speed;
  private String kind;
  ...
}
```

A **`Computer`** has only one **`Memory`**

But neither *is-a* the other

Initializing Data Fields in a Subclass

- What about data fields of a superclass?
  - Initialize them by invoking a superclass constructor with the appropriate parameters
- If the subclass constructor skips calling the superclass ...
  - Java automatically calls the *no-parameter* one
- ***Point:*** Insure superclass fields initialized *before* subclass starts to initialize its part of the object

Example of Initializing Subclass Data

```
public class Computer {
  private String manufacturer;  ...
  public Computer (String manufacturer, ...) {
    this.manufacturer = manufacturer; ...
  }
}

public class Laptop extends Computer {
  private double weight; ...
  public Laptop (String manufacturer, ...,
                 double weight, ...) {
    super(manufacturer, ...);
    this.weight = weight;
  }
}
```


Method *Overriding*

- If subclass has a method of a superclass (same signature), that method *overrides* the superclass method:

```
public class A { ...
    public int M (float f, String s) { bodyA }
}

public class B extends A { ...
    public int M (float f, String s) { bodyB }
}
```

- If we call **M** on an instance of **B** (or subclass of **B**), **bodyB** runs
- In **B** we can access **bodyA** with: **super.M(...)**
- The subclass **M** must have same return type as superclass **M**

Rules for Overriding

- Arguments must be the same, and return types must be compatible.
- The method cannot be less accessible.
  - Public → Protected → Private (not allowed)
  - Private → Protected → Public

Method *Overloading*

- **Method overloading:** *multiple* methods ...
  - With the *same name*
  - But *different signatures*
  - In the *same class*
- Constructors are often overloaded
- Example:
  - `MyClass (int inputA, int inputB)`
  - `MyClass (float inputA, float inputB)`

Example of Overloaded Constructors

```
public class Laptop extends Computer {
  private double weight; ...
  public Laptop (String manufacturer,
                 String processor, ...,
                 double weight, ...) {
    super(manufacturer, processor, ...);
    this.weight = weight;
  }

public Laptop (String manufacturer, ...,
                 double weight, ...) {
    this(manufacturer, "Pentium", ...,
         weight, ...);
  }
}
```

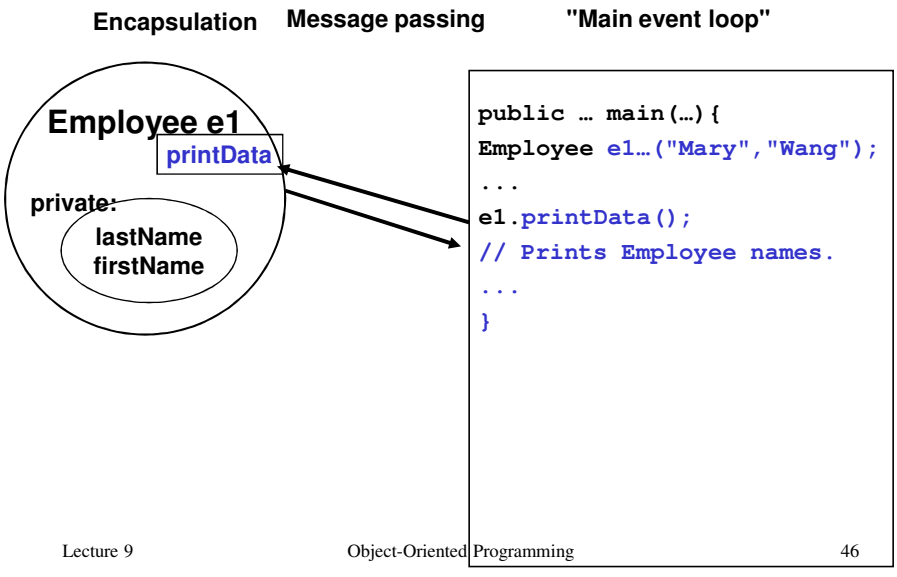

Rules of Overloading

- The return types can be different..
- You can vary the access levels in any direction.

Programming Example

- **A Company has a list of Employees. It asks you to provide a payroll sheet for all employees.**
  - Has extensive data (name, department, pay amount, …) for all employees.
  - Different types of employees – manager, engineer, software engineer.
  - You have an old Employee class but need to add very different data and methods for managers and engineers.
    - Suppose someone wrote a name system, and already provided a legacy Employee class. The old Employee class had a printData() method for each Employee that only printed the name. We want to reuse it, and print pay info.

REVIEW PICTURE



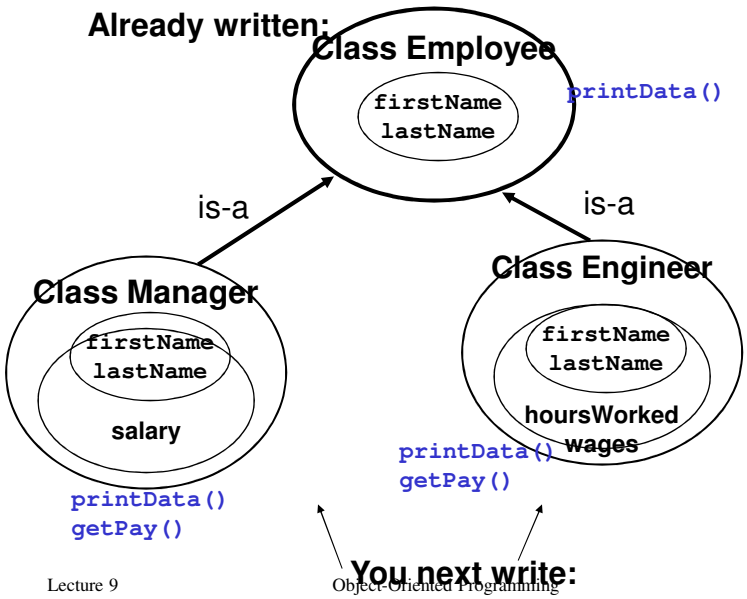
Employee class

This is a simple super or base class.

```
class Employee {
    // Data
    private String firstName, lastName;

    // Constructor
    public Employee(String fName, String lName) {
        firstName= fName; lastName= lName;
    }
    // Method
    public void printData() {
        System.out.println(firstName + " " + lastName);}
}
```


Inheritance



Engineer class

Subclass or (directly) derived class

```
class Engineer extends Employee {
    private double wage;
    private double hoursWorked;
    public Engineer(String fName, String lName,
                    double rate, double hours) {
        super(fName, lName);
        wage = rate;
        hoursWorked = hours;
    }
    public double getPay() {
        return wage * hoursWorked;
    }
    public void printData() {
        super.printData();     // PRINT NAME
        System.out.println("Weekly pay: $" + getPay(); }
}
```


Manager class

Subclass or (directly) derived class

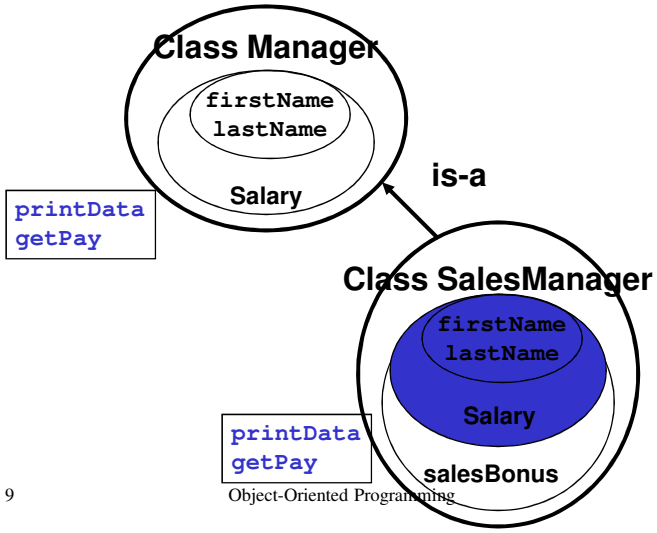
```
class Manager extends Employee {
    private double salary;

    public Manager(String fName, String lName, double sal){
        super(fName, lName);
        salary = sal; }

    public double getPay() {
        return salary; }

    public void printData() {
        super.printData();
        System.out.println("Monthly salary: $" + salary);}
}
```

Inheritance…



SalesManager Class

(Derived class from derived class)

```
class SalesManager extends Manager {
   private double bonus;      // Bonus Possible as commission.

   // A SalesManager gets a constant salary of $1250.0
   public SalesManager(String fName, String lName, double b) {
       super(fName, lName, 1250.0);
       bonus = b; }

   public double getPay() {
       return 1250.0; }

   public void printData() {
       super.printData();
       System.out.println("Bonus Pay: $" + bonus; }
}
```

Main method

```
public class PayRoll {
 public static void main(String[] args) {
   // Could get Data from tables in a Database.
   Engineer fred   = new Engineer("Fred", "Smith", 12.0, 8.0);
   Manager ann      = new Manager("Ann", "Brown", 1500.0);
   SalesManager mary= new SalesManager("Mary", "Kate", 2000.0);


   // Polymorphism, or late binding
   Employee[] employees = new Employee[3];
   employees[0]= fred;
   employees[1]= ann;
   employees[2]= mary;
   for (int i=0; i < 3; i++)
      employees[i].printData();
    }
}
```

Java knows the object type and chooses the appropriate method at run time

Output from main method

```
Fred Smith
Weekly pay: $96.0
Ann Brown
Monthly salary: $1500.0
Mary Barrett
Monthly salary: $1250.0
Bonus: $2000.0
```

Note that we could not write:

```
employees[i].getPay();
```

because `getPay()` is not a method of the superclass Employee.

In contrast, `printData()` is a method of Employee, so Java can find the appropriate version.

Readings

- **Book Name:** An object Oriented Programming with JAVA

  Author Name: C Thomas WU

  Content: Chapter # 13

Acknowledgements

- While preparing this course I have greatly benefited from the material developed by the following people:
  - Aaron Bloomfield (University of Virginia)
  - Andy Van Dam (Brown University)
  - C Thomas Wu (Naval Postgraduate School)
  - MIT-AITI Kenya