



Object-Oriented Programming Concept

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



Outline

- ☐ Part I – Encapsulation
- ☐ Part II – Call by Reference
- ☐ Part III – Inheritance
- ☐ Part IV – Abstract Class and Interface
- ☐ Part V – Polymorphism



OOP Concept Part I

Encapsulation

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



Introduction

- ❑ Classes are the most important language feature that make *object-oriented programming (OOP)* possible
- ❑ Programming in Java consists of defining a number of classes
 - Every program is a class
 - All helping software consists of classes
 - All programmer-defined types are classes
- ❑ Classes are central to Java



A Class Is a Type

- ❑ A class is a special kind of programmer-defined type, and variables can be declared of a class type
- ❑ A value of a class type is called an object or *an instance of the class*
 - If Cat is a class, then the phrases “cat is of type Cat,” “cat is an object of the class Cat,” and “cat is an instance of the class Cat” mean the same thing
- ❑ A class determines the types of data that an object can contain, as well as the actions it can perform



Primitive Type Values vs. Class Type Values

- ❑ A primitive type value is a **single piece of data**
- ❑ A class type value or object can have **multiple pieces of data**, as well as actions called *methods*
 - All objects of a class have the same methods
 - All objects of a class have the same pieces of data (i.e., name, type, and number)
 - For a given object, each piece of data can hold a different value



The Contents of a Class Definition

- ❑ A class definition specifies the data items and methods that all of its objects will have
- ❑ These data items and methods are sometimes called *members* of the object
- ❑ **Data items are called *fields* or *instance variables***
- ❑ Instance variable declarations and method definitions can be placed in any order within the class definition



Lab

```
public class Duck {  
  
    public boolean canfly = false;  
  
    public void quack(){  
        System.out.println("Quack!!");  
    }  
  
}
```




The new Operator

- ❑ An object of a class is named or declared by a variable of the class type:

```
ClassName classVar;
```

- ❑ The **new** operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

- ❑ These can be combined as follows:

```
ClassName classVar = new ClassName();
```



Lab

```
public class Farm {  
  
    public static void main(String[] args) {  
        Duck duck = new Duck();  
    }  
  
}
```



Instance Variables and Methods

- ❑ Instance variables can be defined as in the following two examples
 - Note the **public** modifier (for now):
public String instanceVar1;
public int instanceVar2;
- ❑ In order to refer to a particular instance variable, preface it with its object name as follows:
objectName.instanceVar1
objectName.instanceVar2



Instance Variables and Methods

- ❑ Method definitions are divided into two parts: a *heading* and a *method body*:

```
public void myMethod() ← Heading  
{  
    code to perform some action  
    and/or compute a value  
}
```

Body

- ❑ Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- ❑ Invoking a method is equivalent to executing the method body



Lab

```
public class Farm {  
  
    public static void main(String[] args) {  
  
        Duck duck = new Duck();  
  
        boolean canTheDuckFly = duck.canfly;  
        if(canTheDuckFly == true){  
            System.out.println("The duck can fly");  
        }  
  
        duck.quack();  
    }  
}
```



Local Variables

- ❑ A variable declared within a method definition is called a *local variable*
 - All variables declared in a method are local variables
 - All method parameters are local variables
- ❑ If two methods each have a local variable of the same name, they are still two entirely different variables

```
public class Duck {  
    public boolean canfly = false;  
  
    public String eat(String food){  
        String message = "Thank you! The " + food + " is good!";  
        return message;  
    }  
}
```

instance variable

local variable



Parameters of a Method

- ❑ A parameter list provides a description of the data required by a method

```
public double myMethod(int p1, int p2, double p3) {  
    double sum = p1 + p2 + p3;  
    return sum;  
}
```



Arguments

- ❑ When a method is invoked, the type of each argument must be compatible with the type of the corresponding parameter

```
int    a=1;  
int    b=2;  
double c=3.0;  
double result = myMethod(a,b,c);
```




Automatic Upper Casting

- ❑ A primitive argument can be automatically type cast from any of the following types, to any of the types that appear to its right:

`byte`→`short`→`int`→`long`→`float`→`double`
`Char` —————↑

For example:

```
int    a=1;  
int    b=2;  
int    c=3;  
double result = myMethod(a,b,c);
```



Encapsulation

- ❑ *Encapsulation* means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
 - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
 - In Java, hiding details is done by marking them **private**



public and private Modifiers

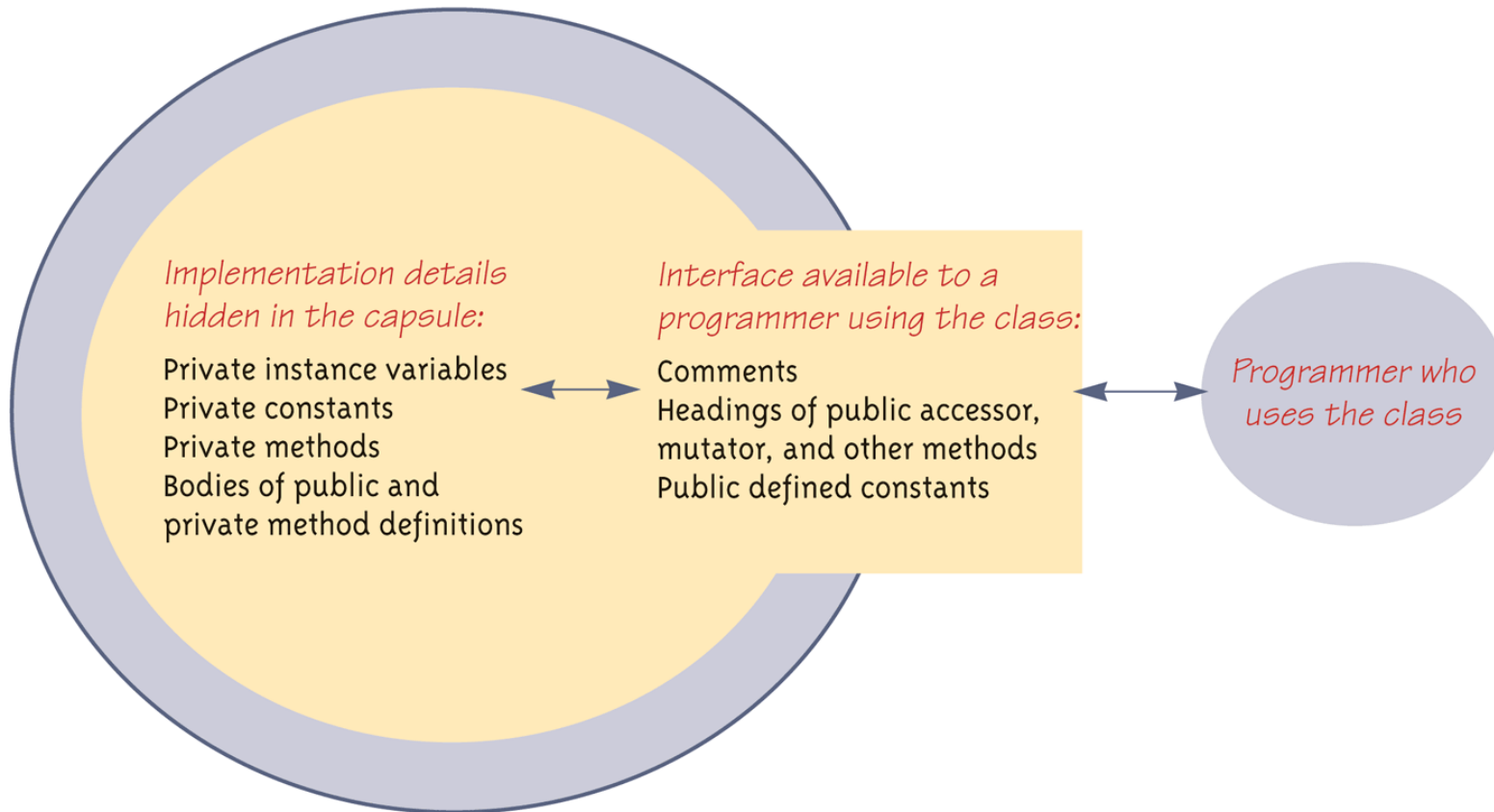
- ❑ The modifier **public** means that there are no restrictions on where an instance variable or method can be used
- ❑ The modifier **private** means that an instance variable or method cannot be accessed by name outside of the class
 - It is considered good programming practice to make **all** instance variables **private**
 - Most methods are **public**, and thus provide controlled access to the object
 - Usually, methods are **private** only if used as helping methods for other methods in the class



Encapsulation

Display 4.10 Encapsulation

An encapsulated class



A class definition should have no public instance variables.



Lab

```
public class Duck {  
  
    private boolean canfly = false;  
  
    public boolean getCanfly(){  
        return canfly;  
    }  
  
    ...  
}
```



Lab

```
public class Farm {  
  
    public static void main(String[] args) {  
        Duck duck = new Duck(true);  
  
        boolean canTheDuckFly = duck.getCanfly();  
        if(canTheDuckFly == true){  
            System.out.println("The duck can fly");  
        }  
  
        ...  
    }  
  
}
```



Overloading

- ❑ *Overloading* is when two or more methods *in the same class* have the same method name
- ❑ To be valid, any two definitions of the method name must have different *signatures*
 - A signature consists of the name of a method together with its parameter list
 - Differing signatures must have different numbers and/or types of parameters



Lab

```
public class Duck {
    ...
    public void quack(){
        System.out.println("Quack!!");
    }
    public void quack(String sound){
        System.out.println(sound);
    }
    ...
}

public class Farm {

    public static void main(String[] args) {
        Duck duck = new Duck(true);
        ...
        duck.quack();
        duck.quack("Ga Ga Ga");
    }
}
```




OOP Concept Part II

Call by Reference

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



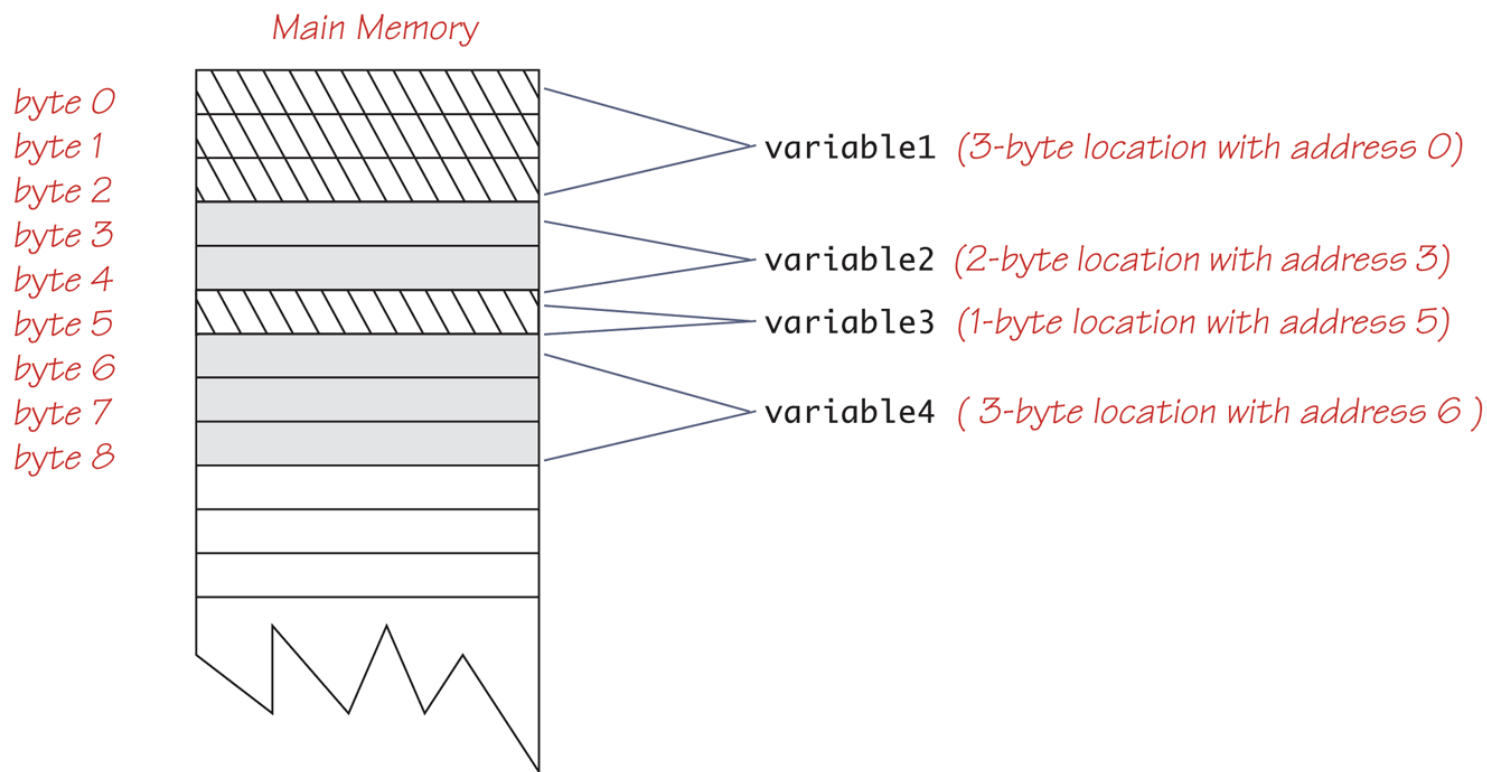
Variables and Memory

- ❑ Values of most data types require more than one byte of storage
 - Several **adjacent bytes** are then used to hold the data item
 - The entire **chunk of memory** that holds the data is called its *memory location*
 - The address of the **first byte** of this memory location is used as the **address** for the data item
- ❑ A computer's main memory can be thought of as a long list of memory locations of *varying sizes*



Variables in Memory

Display 5.10 **Variables in Memory**





References

- ❑ Every variable is implemented as a location in computer memory
- ❑ When the variable is a **primitive type**, the **value of the variable is stored in the memory location** assigned to the variable
 - Each primitive type always require the same amount of memory to store its values



References

- ❑ When the variable is **a class type**, **only the memory address (or *reference*)** where its object is located is stored in the memory location assigned to the variable
 - The object named by the variable is stored in some other location in memory
 - Like primitives, the value of a class variable is a fixed size
 - Unlike primitives, the value of a class variable is a memory address or reference
 - The object, whose address is stored in the variable, can be of any size



Class Type Variables Store a Reference (Part 1 of 2)

Display 5.12 Class Type Variables Store a Reference

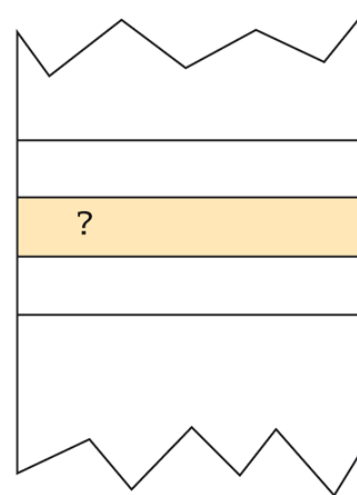
```
public class ToyClass
{
    private String name;
    private int number;
```

*The complete definition of the class
ToyClass is given in Display 5.11.*

```
ToyClass sampleVariable;
```

*Creates the variable **sampleVariable** in
memory but assigns it no value.*

sampleVariable



```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

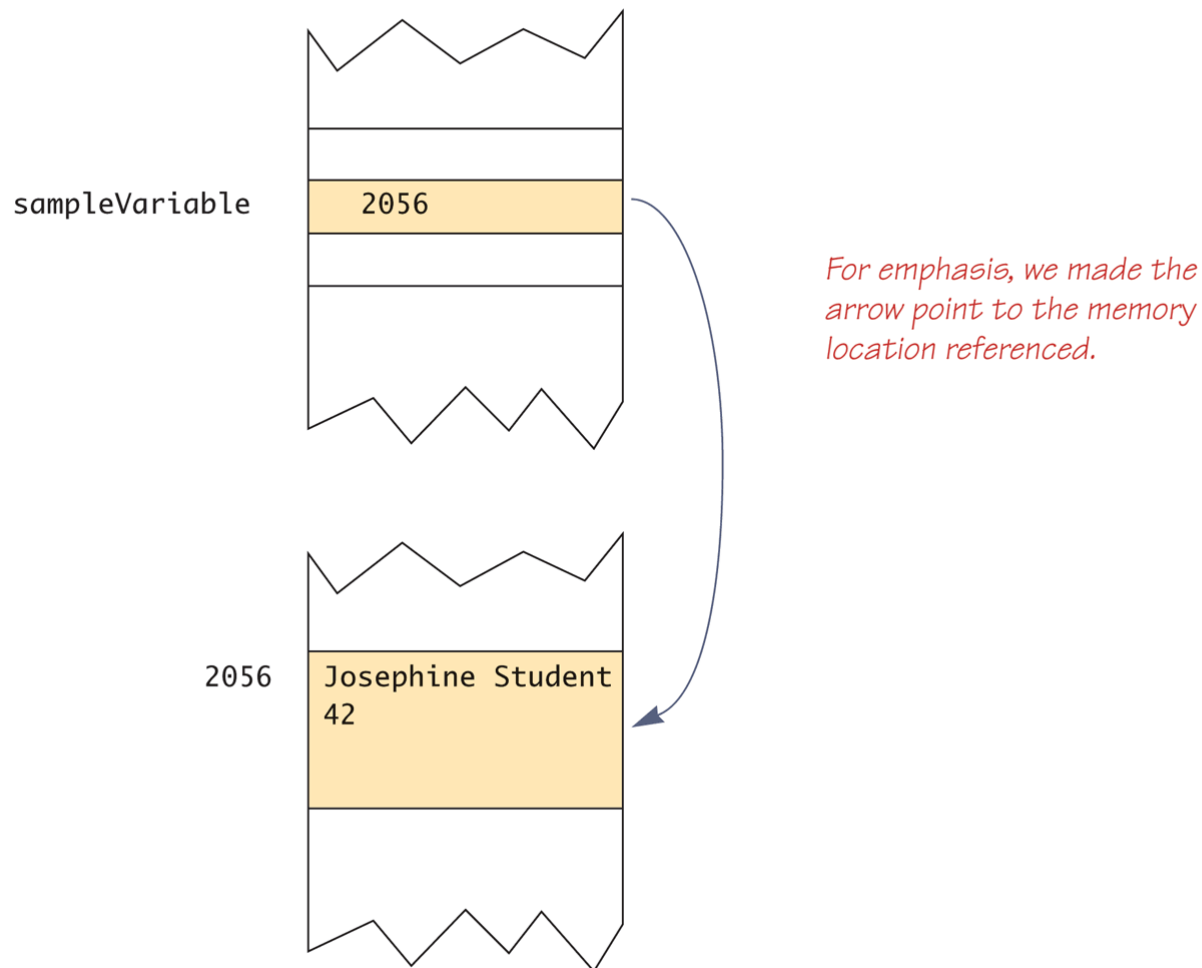
*Creates an object, places the object someplace in memory, and then
places the address of the object in the variable **sampleVariable**. We
do not know what the address of the object is, but let's assume it is
2056. The exact number does not matter.*

(continued)



Class Type Variables Store a Reference (Part 2 of 2)

Display 5.12 Class Type Variables Store a Reference





References

- ❑ **Two reference variables can contain the same reference, and therefore name the same object**
 - The assignment operator sets the reference (memory address) of one class type variable equal to that of another
 - Any change to the object named by one of these variables will produce a change to the object named by the other variable, since they are the same object

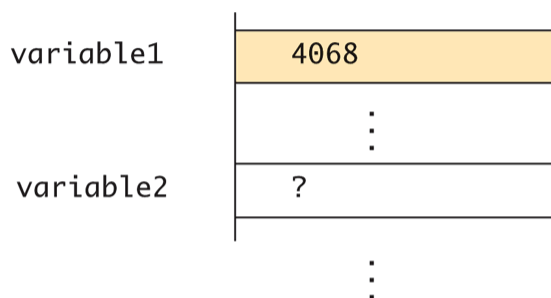
```
variable2 = variable1;
```




Assignment Operator with Class Type Variables (Part 1 of 3)

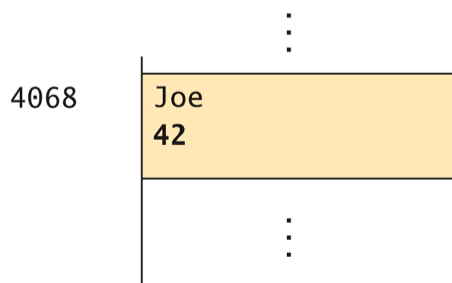
Display 5.13 Assignment Operator with Class Type Variables

```
ToyClass variable1 = new ToyClass("Joe", 42);  
ToyClass variable2;
```



*We do not know what memory address (reference) is stored in the variable **variable1**. Let's say it is 4068. The exact number does not matter.*

Someplace else in memory:



Note that you can think of

```
new ToyClass("Joe", 42)
```

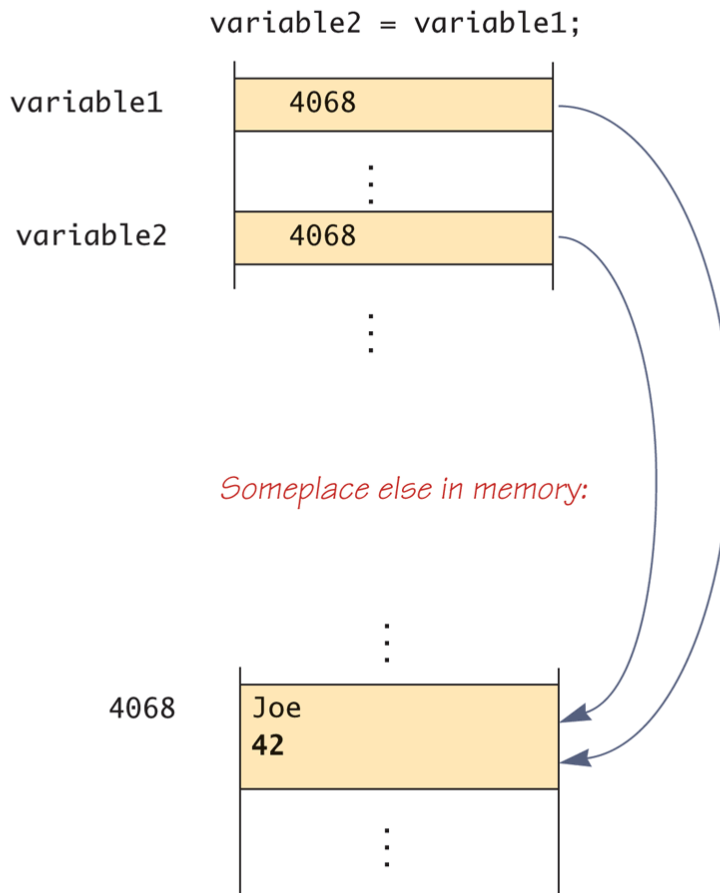
as returning a reference.

(continued)



Assignment Operator with Class Type Variables (Part 2 of 3)

Display 5.13 Assignment Operator with Class Type Variables



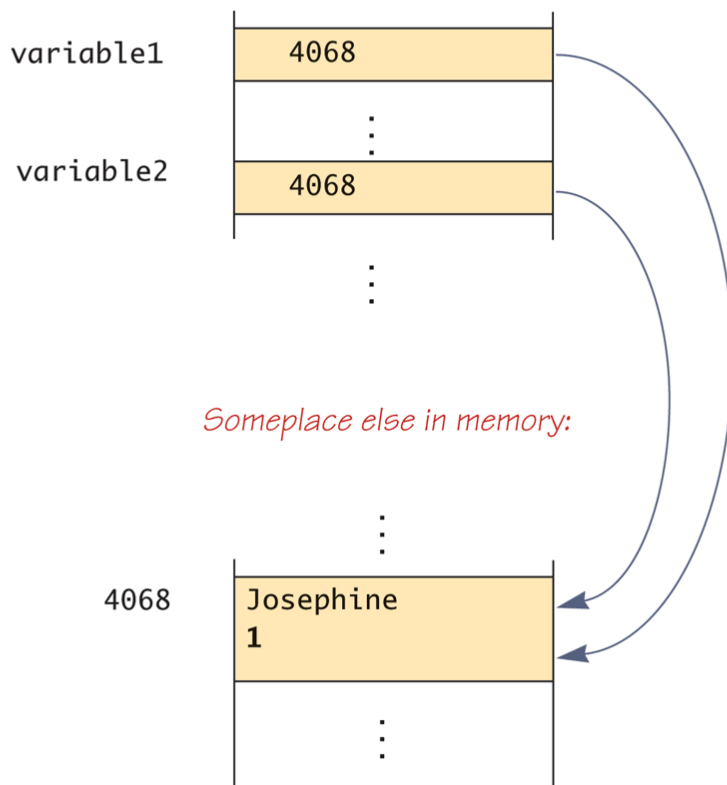
(continued)



Assignment Operator with Class Type Variables (Part 3 of 3)

Display 5.13 Assignment Operator with Class Type Variables

```
variable2.set("Josephine", 1);
```





Lab

```
public class Cat {  
  
    int age = 1;  
  
    public static void main(String[] args)  
    {  
        Cat cat1 = new Cat();  
        Cat cat2 = cat1;  
  
        cat1.age = 2;  
        System.out.println(cat2.age);  
    }  
}
```



Class Parameters

- ❑ **Primitive type parameters** in Java are *call-by-value* parameters
 - A parameter is a *local variable* that is set equal to the value of its argument
 - Therefore, any change to the value of the parameter cannot change the value of its argument
- ❑ **Class type parameters** appear to behave differently from primitive type parameters
 - They appear to behave in a way similar to parameters in languages that have the *call-by-reference* parameter passing mechanism



Differences Between Primitive and Class-Type Parameters

- ☐ A method **cannot change the value of a variable of a primitive type** that is an argument to the method
- ☐ In contrast, a method **can change the values of the instance variables of a class type** that is an argument to the method



Lab (Call by Value)

```
public class PrimitiveParameterDemo {  
  
    public static void main(String[] args)  
    {  
        int speed = 50;  
        System.out.println("argument value:" + speed);  
  
        changer(speed);  
  
        System.out.println("argument value:" + speed);  
    }  
  
    public static void changer(int speed)  
    {  
        speed = 100;  
        System.out.println("parameter value:" + speed);  
    }  
}
```



Lab (Call by Reference)

```
public class ToyClass
{
    private String name;
    private int number;

    public ToyClass(String initialName, int initialNumber)
    {
        name = initialName;
        number = initialNumber;
    }

    public String toString( )
    {
        return (name + " " + number);
    }

    public void set(String newName, int newNumber)
    {
        name = newName;
        number = newNumber;
    }
}
```




Lab (Call by Reference)

```
public class ClassParameterDemo
{
    public static void main(String[] args)
    {
        ToyClass anObject = new ToyClass("Robot Dog", 10);
        System.out.println(anObject);

        changer(anObject);
        System.out.println(anObject);
    }

    public static void changer(ToyClass aParameter)
    {
        aParameter.set("Robot Cat", 20);
    }
}
```



OOP Concept Part III

Inheritance

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



Inheritance

- ❑ 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

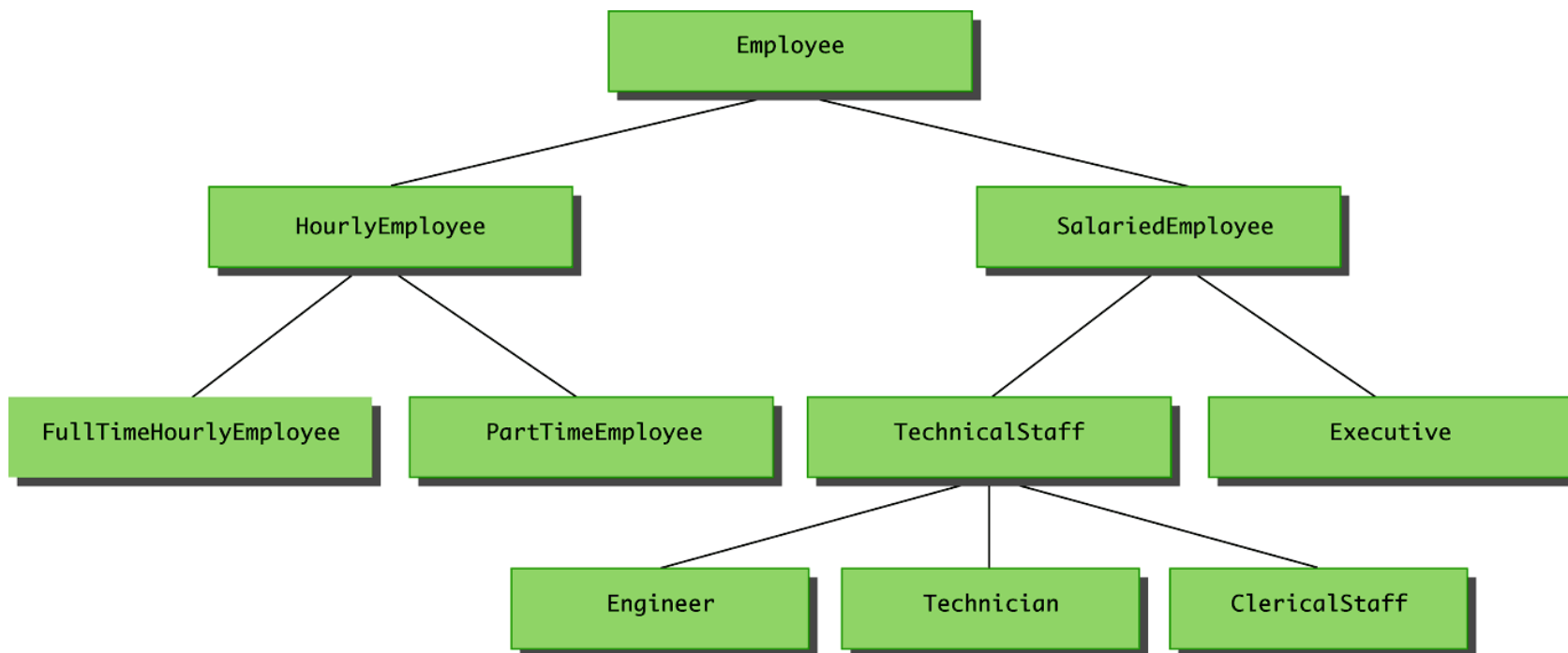
- ❑ Within Java, a class called **Employee** can be defined that includes all employees
- ❑ This class can then be used to define classes for hourly employees and salaried employees
 - In turn, the **HourlyEmployee** class can be used to define a **PartTimeHourlyEmployee** class, and so forth

```
public class HourlyEmployee extends Employee
```



A Class Hierarchy

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.



Lab

```
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;
    }
}
```




Lab

```
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;
    }
}
```



Lab

```
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



Lab

```
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!



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**



OOP Concept Part IV

Abstract Class & Interface

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



Introduction to Abstract Classes

- ❑ In order to postpone the definition of a method, Java allows an *abstract method* to be declared
 - An abstract method has a heading, but no method body
 - The body of the method is defined in the derived classes
- ❑ The class that contains an abstract method is called an *abstract class*



Abstract Method

- ❑ An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- ❑ It has a complete method heading, to which has been added the modifier **abstract**
- ❑ **It cannot be private**
- ❑ It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPay();  
public abstract void doIt(int count);
```



Abstract Class

- ❑ A class that has at least one abstract method is called an *abstract class*
 - An abstract class must have the modifier **abstract** included in its class heading:

```
public abstract class Employee
{
    private instanceVariables;
    . . .
    public abstract double getPay();
    . . .
}
```



Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add **abstract** to its modifier
- ❑ A class that has no abstract methods is called a *concrete class*



Pitfall: You Cannot Create Instances of an Abstract Class

- ❑ An abstract class can only be used to derive more specialized classes
 - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- ❑ **An abstract class constructor cannot be used to create an object of the abstract class**



Lab

```
public abstract class Animal {  
    public abstract void run();  
    public void sit(){ System.out.println("Sit down..."); }  
}
```

```
public class Dog extends Animal {  
    public void run(){  
        System.out.println("The dog is running");  
    }  
}
```

```
public class Cat extends Animal{  
    public void run(){  
        System.out.println("The cat is running");  
    }  
}
```



Lab

```
public class House {  
  
    public static void main(String[] args) {  
        Animal dog = new Dog();  
        Animal cat = new Cat();  
  
        playWith(dog);  
        playWith(cat);  
  
        dog.sit();  
        cat.sit();  
    }  
  
    public static void playWith(Animal animal){  
        animal.run();  
    }  
}
```



Interfaces

- ❑ An *interface* is something like an extreme case of an abstract class
 - However, *an interface is not a class*
 - *It is a type that can be satisfied by any class that implements the interface*
- ❑ The syntax for defining an interface is similar to that of defining a class
 - Except the word **interface** is used in place of **class**



Interfaces

- ❑ An interface specifies a set of methods that any class that implements the interface must have
 - It contains **method headings** and **constant definitions** only
 - Any variables defined in an interface must be public, static, and final
 - It contains **no instance variables nor any complete method definitions**



Lab (Constants)

```
public interface Shape {  
    int color = 1; // => public static final int color = 1;  
}
```

```
public class Paint {  
    public static void main(String[] args) {  
        System.out.println(Shape.color);  
    }  
}
```



Interfaces

- ❑ All methods in an interface are **implicitly public and abstract**, so you can omit the public modifier.
 - They cannot be given private or protected

```
public interface ISpec1 {  
    private void run(); //Not allowed  
    protected void run(); //Not allowed  
  
    void run(); // Allowed. Equal to the following definition  
    public abstract void run(); //Allowed  
  
}
```



Lab

```
public interface Shape {  
    int color = 1; // => public static final int color = 1;  
    public abstract double area(); //=> double area();  
}
```



Interfaces

- ❑ *Multiple inheritance* is not allowed in Java
- ❑ Instead, Java's way of approximating multiple inheritance is through interfaces

```
public class ConcreteClass implements ISpec1, ISpec2, ISpec3{  
    ...  
}
```



Interfaces

- ❑ To *implement an interface*, a concrete class must do two things:
 1. `implements Interface_Name`
 2. The class must implement *all* the method headings listed in the definition(s) of the interface(s)



Lab

```
public class Rectangle implements Shape{  
    int x1=0;  
    int y1=0;  
    int x2=10;  
    int y2=10;  
    public double area(){  
        return (x2-x1)*(y2-y1);  
    }  
}
```

```
public class Circle implements Shape{  
  
    double radius = 3;  
    public double area(){  
        return radius*radius*3.14;  
    }  
}
```



Lab

```
public class Paint {  
  
    public static void main(String[] args) {  
        System.out.println(Shape.color);  
  
        Shape shape1 = new Rectangle();  
        printArea(shape1);  
  
        Shape shape2 = new Circle();  
        printArea(shape2);  
    }  
  
    public static void printArea(Shape shape){  
        System.out.println(shape.area());  
    }  
}
```



Abstract Classes Implementing Interfaces

- ❑ Abstract classes may implement one or more interfaces
 - Any method headings given in the interface that are not given definitions are made into abstract methods
- ❑ A concrete class must give definitions for all the method headings given in the abstract class *and the interface*



Abstract Class vs. Interface

```
public abstract class Animal {  
    public abstract void run();  
    public void sit(){ System.out.println("Sit down..."); }  
}
```

vs.

```
public interface Shape {  
  
    int color = 1; // => public static final int color = 1;  
  
    public abstract double area(); //=> double area();  
}
```



Derived Interfaces

- ❑ Like classes, an interface may be derived from a base interface
 - This is called *extending* the interface
 - The derived interface must include the phrase
`extends BaseInterfaceName`
- ❑ A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface



Lab

```
public interface Drawing {  
    public abstract void drawBorder();  
}
```

```
public interface Shape extends Drawing{
```

```
    int color = 1; // => public static final int color = 1;
```

```
    public abstract double area();  
}
```



Lab

```
public class Rectangle implements Shape{
```

```
    int x1=0;
```

```
    int y1=0;
```

```
    int x2=10;
```

```
    int y2=10;
```

```
    public double area(){
```

```
        return (x2-x1)*(y2-y1);
```

```
    }
```

```
    public void drawBorder(){
```

```
        System.out.println("Drawing the border of the rectangle...");
```

```
    }
```

```
}
```



Lab

```
public class Circle implements Shape{
```

```
    double radius = 3;  
    public double area(){  
        return radius*radius*3.14;  
    }
```

```
    public void drawBorder(){  
        System.out.println("Drawing the border of the circle...");  
    }  
}
```



OOP Concept Part V

Polymorphism

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



Late Binding

❑ *Early binding or static binding*

- **which method is to be called** is decided at compile-time
 - *Overloading*: an invocation can be operated on arguments of more than one type

❑ *Late binding or dynamic binding*

- **which method is to be called** is decided at compile-time
 - *Overriding*: a derived class inherits methods from the base class, it can change or override an inherited method



Lab: Early binding (through overloading)

```
public class SayHello {

    public String sayHello(String name){
        return "Hello! "+ name;
    }

    public String sayHello(String name, String gender){
        if(gender.equals("boy")){
            return "Hello! Mr. "+ name;
        }
        else if(gender.equals("girl")){
            return "Hello! Miss. "+ name;
        }else{
            return "Hello! "+ name;
        }
    }

    public static void main(String[] args){
        SayHello hello = new SayHello();
        System.out.println(hello.sayHello("S.J.)); //decided at compile time
        System.out.println(hello.sayHello("S.J.", "boy")); //decided at compile time
    }
}
```




Lab: Late binding (through overriding)

```
public class Payment {  
    public void pay(){  
        System.out.println("Pay in cash");  
    }  
    public void checkout(){  
        pay();  
    }  
}
```

```
public class Store {  
    public static void main(String[] args) {  
        Payment p1 = new Payment();  
        p1.checkout();  
    }  
}
```



Lab: Late binding (through overriding)

```
public class CreditCardPayment extends Payment{  
    public void pay() {  
        System.out.println("Pay with credit card");  
    }  
}
```

```
public class Store {  
    public static void main(String[] args) {  
        Payment p1 = new Payment();  
        p1.checkout();  
  
        Payment p2 = new CreditCardPayment();  
        p2.checkout();  
    }  
}
```



Pitfall: No Late Binding for Static Methods

- ❑ Java uses **static binding** with **private**, **final**, and **static** methods
 - In the case of **private** and **final** methods, late binding would serve no purpose
 - However, in the case of a static method invoked using a calling object, it does make a difference



Lab

```
public class Payment {  
    public static void pay(){  
        System.out.println("Pay in cash");  
    }  
    public void checkout(){  
        pay();  
    }  
}
```

```
public class CreditCardPayment extends Payment{  
    public static void pay() {  
        System.out.println("Pay with credit card");  
    }  
}
```

Then run Store again!



Lab

```
public class Store {  
    public static void main(String[] args) {  
        Payment p1 = new Payment();  
        p1.checkout();  
  
        Payment p2 = new CreditCardPayment();  
        p2.checkout();  
    }  
}
```

the type of **p2** is determined by its variable name, not the object that it references



Upcasting and Downcasting

- ❑ *Upcasting* is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)

```
Payment p2 = new CreditCardPayment();  
p2.checkout();
```



Upcasting and Downcasting

- ❑ *Downcasting* is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)
 - Downcasting has to be done very carefully
 - In many cases it doesn't make sense, or is illegal:

```
Payment p1 = new Payment();  
CreditCardPayment p2 = (CreditCardPayment)p1; //runtime error
```



Tip: Checking to See if Downcasting is Legitimate

- ❑ Downcasting to a specific type is only sensible if the object being cast is an instance of that type
 - This is exactly what the **instanceof** operator tests for:
object instanceof ClassName
 - It will return true if ***object*** is of type ***ClassName***
 - In particular, it will return true if ***object*** is an instance of any descendent class of ***ClassName***



Lab (Downcasting)

Step1: Remove “static” in `CreditCardPayment` and `Payment`

Step2

```
public class CreditCardPayment extends Payment{  
    public void pay() {  
        System.out.println("Pay with credit card");  
    }  
    public void sign(){  
        System.out.println("Signing...");  
    }  
}
```



Lab (Downcasting)

```
public class Store {  
  
    public static void main(String[] args) {  
        Payment p1 = new Payment();  
        p1.checkout();  
        payProcess(p1);  
  
        Payment p2 = new CreditCardPayment();  
        p2.checkout();  
        payProcess(p2);  
    }  
  
    public static void payProcess(Payment p){  
        if(p instanceof CreditCardPayment){  
            ((CreditCardPayment)p).sign();  
        }  
    }  
}
```