Chulalongkorn University
International School of Engineering
Department of Computer Engineering
2140105 Computer Programming Lab.

Name _____
Student ID _____
Station No. _____
Date _____

Lab 10 – OO Concept (Episode IV)

Objectives:

- Understand inheritance
- Be able to use inheritance for reusability

Inheritance

In the preceding lessons, you have seen inheritance mentioned several times. In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.

Definitions:

> A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Excepting `Object`, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, *Object*. Such a class is said to be descended from all the classes in the inheritance chain stretching back to Object.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass *inherits* all the non-private members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The Java Platform Class Hierarchy

The `Object` class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

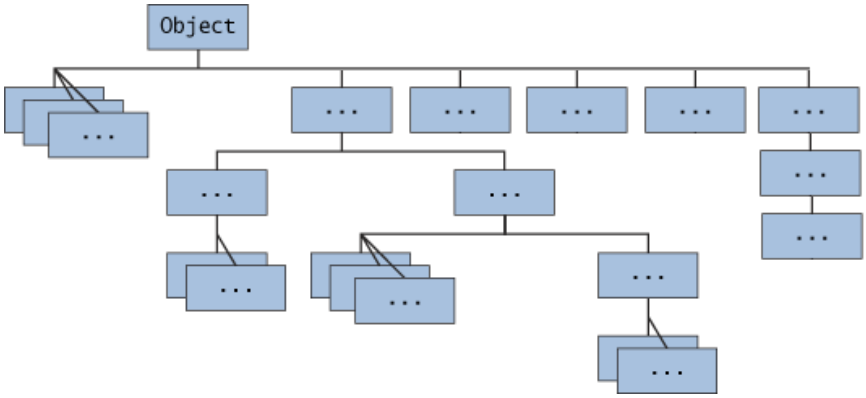

Figure 1 All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, `Object` is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

An Example of Inheritance

Here is the sample code for a possible implementation of a `Bicycle` class that was presented in Lab 8:

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a `MountainBike` class that is a subclass of `Bicycle` might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

`MountainBike` inherits all the fields and methods of `Bicycle` and adds the field `seatHeight` and a method to set it. Except for the constructor, it is as if you had written a new `MountainBike` class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the `Bicycle` class were complex and had taken substantial time to debug.

What You Can Do in a Subclass

A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the package-private members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus ***hiding*** it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus ***overriding*** it.
- You can write a new static method in the subclass that has the same signature as the one in the superclass, thus ***hiding*** it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

Private Members in a Superclass

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

```
public MountainBike myBike = new MountainBike();
```

then `myBike` is of type `MountainBike`.

`MountainBike` is descended from `Bicycle` and `Object`. Therefore, a `MountainBike` is a `Bicycle` and is also an `Object`, and it can be used wherever `Bicycle` or `Object` objects are called for.

The reverse is not necessarily true: a `Bicycle` may be a `MountainBike`, but it isn't necessarily. Similarly, an `Object` may be a `Bicycle` or a `MountainBike`, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
```

then `obj` is both an `Object` and a `Mountainbike` (until such time as `obj` is assigned another object that is not a `Mountainbike`). This is called **implicit casting**.

If, on the other hand, we write

```
MountainBike myBike = obj;
```

we would get a compile-time error because `obj` is not known to the compiler to be a `MountainBike`. However, we can tell the compiler that we promise to assign a `MountainBike` to `obj` by explicit casting:

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that `obj` is assigned a `MountainBike` so that the compiler can safely assume that `obj` is a `MountainBike`. If `obj` is not a `Mountainbike` at runtime, an exception will be thrown.

Note: You can make a logical test as to the type of a particular object using the `instanceof` operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {
   MountainBike myBike = (MountainBike)obj;
}
```

Here the `instanceof` operator verifies that `obj` refers to a `MountainBike` so that we can make the cast with knowledge that there will be no runtime exception thrown.

Overriding and Hiding Methods

Instance Methods

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The *overriding* method has the same name, number and type of parameters, and return type as the method it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This is called a covariant return type.

Class Methods

If a subclass defines a class method with the same signature as a class method in the superclass, the method in the subclass *hides* the one in the superclass.

The distinction between hiding and overriding has important implications. The version of the overridden method that gets invoked is the one in the subclass. The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass. Let's look at an example that contains two classes. The first is Animal, which contains one instance method and one class method:

Your turn (1)

1. Create a new Java project called `lab10`.
2. Create a new package called `inheritance`.
3. Create a new class called `Animal` in package `inheritance` and copy the following code:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Animal.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}
```

4. Create a new class as a subclass of `Animal` called `Cat` in package `inheritance` and copy the follwing code:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Cat.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat.");
```

```
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

The `Cat` class overrides the instance method in `Animal` and hides the class method in `Animal`. The main method in this class creates an instance of `Cat` and calls `testClassMethod()` on the class and `testInstanceMethod()` on the instance.

5. Run the Cat class and see the output, compare to the following text:

```
The class method in Animal.

The instance method in Cat.
```

As promised, the version of the hidden method that gets invoked is the one in the superclass, and the version of the overridden method that gets invoked is the one in the subclass.

Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super, which is covered in the next section. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

Using the Keyword super

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super. You can also use super to refer to a hidden field (although hiding fields is discouraged).

6. Create a new class called `Superclass` in package inheritance:

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

7. Create a subclass called `Subclass`, that overrides `printMethod()`:

```
public class Subclass extends Superclass {

    public void printMethod() { //overrides printMethod in Superclass
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
```

```
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Within `Subclass`, the simple name `printMethod()` refers to the one declared in `Subclass`, which overrides the one in `Superclass`. So, to refer to `printMethod()` inherited from `Superclass`, `Subclass` must use a qualified name, using `super` as shown.

8. Run Subclass and see the output compare to the following:

```
Printed in Superclass.

Printed in Subclass
```

Subclass Constructors

The following example illustrates how to use the super keyword to invoke a superclass's constructor. Recall from the `Bicycle` example that `MountainBike` is a subclass of `Bicycle`. Here is the `MountainBike` (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

```
public MountainBike(int startHeight, int startCadence,
                    int startSpeed, int startGear) {
   super(startCadence, startSpeed, startGear);
   seatHeight = startHeight;
}
```

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

```
super();
```

--or--

```
super(parameter list);
```

With `super()`, the superclass no-argument constructor is called. With `super(parameter list)`, the superclass constructor with a matching parameter list is called.

Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. `Object` does have such a constructor, so if `Object` is the only superclass, there is no problem.

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of `Object`. In fact, this is the case. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.

Abstract Methods and Classes

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare non-abstract methods
   abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

Note: All of the methods in an interface (see the Interfaces section) are implicitly abstract, so the abstract modifier is not used with interface methods (it could be—it's just not necessary).

Abstract Classes versus Interfaces

Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains only abstract method declarations, it should be declared as an interface instead.

By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—for example, resize or draw. All `GraphicObjects` must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, `GraphicObject`, as shown Figure 2.

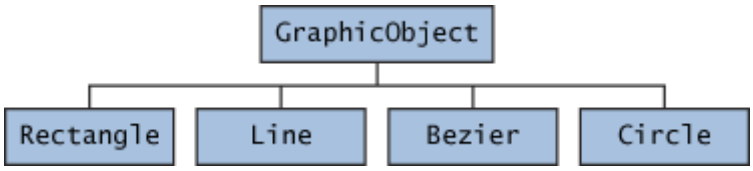

Figure 2 Classes Rectangle, Line, Bezier, and Circle inherit from GraphicObject

First, you declare an abstract class, `GraphicObject`, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the `moveTo` method. `GraphicObject` also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. The `GraphicObject` class can look something like this:

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Each non-abstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, must provide implementations for the draw and resize methods:

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

When an Abstract Class Implements an Interface

In the section on Interfaces , it was noted that a class that implements an interface must implement all of the interface's methods. It is possible, however, to define a class that does not implement all of the interface methods, provided that the class is declared to be abstract. For example,

```
abstract class X implements Y {
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
```

```
}
```

In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

Class Members

An abstract class may have static fields and static methods. You can use these static members with a class reference—for example, `AbstractClass.staticMethod()`—as you would with any other class.

Your turn (2)

1. Create a new package called `exercise2`.
2. Write a class called `Card`, whose instances represent a single playing card from a deck of cards. Playing cards have two distinguishing properties: `rank` and `suit`.
   - rank – 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, and A
   - suit – spades(♠), heart (♥), diamond (♦), and club (♣)
3. Write a class called `Deck`, whose instances represent a **full** deck of cards (52 cards). Hint: This class has a field that is an array of `Card`.
4. Add the following methods to class Deck:
   - shuffle – shuffle the deck
   - pick – pick the top class from a deck
5. Write an application which implements Game interface:

```
public interface Game {
    public void start();  // start the game
    public void stop();   // stop the game
    public void restart();
}
```

   - When start a game,
     - create 2 decks of card, deck1 for computer, and deck2 for you
     - shuffle both decks
     - pick a card from each deck and compare the card. Who has the higher rank get 1 point (2 is the lowest and A is the highest), if the rank is the same, see the suit (spades > heart > diamond > club).
   - When complete deck is picked, print the point and the winner (who has the higher point). Ask the user whether he/she wants to play again or stop. If stop, quit the program. If play again, call the method `restart()` which behaves like `start()` but does not required to create 2 decks.

Some useful methods:

```
/**
* Returns a random number between first and last (exclusive last)
*/
public static int random(int first, int last) {
    return first + (int)(Math.random()*(last – first))
}

/**
* Returns a random number from 0 to N (exclusive N)
*/
public static int random(int n) {
    return random(0, n);
}
```

Lab 10 – OO Concept (Episode IV)

Task	Description	Result	Note
1	Inheritance/Animal		
2	Subclass/Superclass		
3	**`Card`**		
4	**`Deck`**		
5	**`Game`**		
6			
7			