
CS1020 Lecture Note #7:

Object Oriented Programming
Inheritance

Like father, like son

Objectives

- Introducing inheritance through creating subclasses
 - Improve code reusability
 - Allowing overriding to replace the implementation of an inherited method

References



Textbook

- Chapter 1: Section 1.4 (pg 54 – 56)
- Chapter 9: Section 29.1 (pg 480 – 490)



CS1020 website →
Resources → Lectures

- http://www.comp.nus.edu.sg/~cs1020/2_resources/lectures.html

Outline

1. Overriding Methods (revisit)
2. Creating a Subclass
 - 2.1 Observations
 - 2.2 Constructors in Subclass
 - 2.3 The “super” Keyword
 - 2.4 Using SavingAcct
 - 2.5 Method Overriding
 - 2.6 Using “super” Again
3. Subclass Substitutability
4. The “Object” Class
5. “is-a” versus “has-a”
6. Preventing Inheritance (“final”)
7. Constraint of Inheritance in Java
8. Quick Quizzes

0. Object-Oriented Programming

- Four fundamental concepts of OOP:
 - Encapsulation
 - Abstraction
 - **Inheritance**
 - Polymorphism
- **Inheritance** allows new classes to inherit properties of existing classes
- Main concepts in inheritance
 - **Subclassing**
 - **Overriding**

1. Overriding Methods (revisit) (1/2)

- Recall in lecture #4 that a user-defined class automatically inherits some methods – such as `toString()` and `equals()` – from the `Object` class
- The `Object` class is known as the `parent class` (or `superclass`); it specifies some basic behaviours common to all kinds of objects, and hence these behaviours are inherited by all its `subclasses` (`derived classes`)
- However, these inherited methods usually don't work in the subclass as they are not customised

1. Overriding Methods (revisit) (2/2)

- Hence, to make them work, we customised these inherited methods – this is called **overriding**

Lecture #4: MyBall/MyBall.java

```
/****** Overriding methods *****/
// Overriding toString() method
public String toString() {
    return "[" + getColour() + ", " + getRadius() + "];"
}

// Overriding equals() method
public boolean equals(Object obj) {
    if (obj instanceof MyBall) {
        MyBall ball = (MyBall) obj;
        return this.getColour().equals(ball.getColour()) &&
            this.getRadius() == ball.getRadius();
    }
    else
        return false;
}
}
```

2. Creating a Subclass (1/6)

- Object-oriented languages allow **inheritance**
 - Declare a new class based on an existing class
 - So that the new class may inherit all of the attributes and methods from the other class
- Terminology
 - If class *B* is derived from class *A*, then class *B* is called a **child** (or **subclass** or **derived class**) of class *A*
 - Class *A* is called a **parent** (or **superclass**) of class *B*

2. Creating a Subclass (2/6)

- Recall the `BankAcct` class in lecture #4

lect4/BankAcct.java

```
class BankAcct {
    private int acctNum;
    private double balance;

    public BankAcct() { }

    public BankAcct(int aNum, double bal) { ... }

    public int getAcctNum() { ... }


    public double getBalance() {... }

    public boolean withdraw(double amount) { ... }

    public void deposit(double amount) { ... }

    public void print() { ... }
}
```

2. Creating a Subclass (3/6)

- Let's define a **SavingAcct** class
 - Basic information:
 - Account number, balance
 - Interest rate
 - Basic functionality:
 - Withdraw, deposit
 - Pay interest
 - Compare with the basic bank account:
 - Differences are highlighted above
 - SavingAcct shares more than 50% of the code with BankAcct
 - So, should we just cut and paste the code from **BankAcct** to create **SavingAcct**?
- 
- The diagram consists of the text 'New requirements' in red, positioned to the right of the list. Two red arrows originate from this text. One arrow points to the 'Interest rate' item in the 'Basic information' section. The other arrow points to the 'Pay interest' item in the 'Basic functionality' section.

2. Creating a Subclass (4/6)

- Duplicating code is **undesirable** as it is hard to maintain
 - Need to correct all copies if errors are found
 - Need to update all copies if modifications are required
- Since the classes are logically unrelated if the codes are separated:
 - Code that works on one class cannot work on the other
- Compilation errors due to incompatible data types
- Hence, we should create **SavingAcct** as a subclass of **BankAcct**

2. Creating a Subclass (5/6)

BankAcct.java

```
class BankAcct {  
    protected int acctNum;  
    protected double balance;  
  
    //Constructors and methods not shown  
  
}
```

The “protected” keyword allows subclass to access the attributes directly

The “extends” keyword indicates inheritance

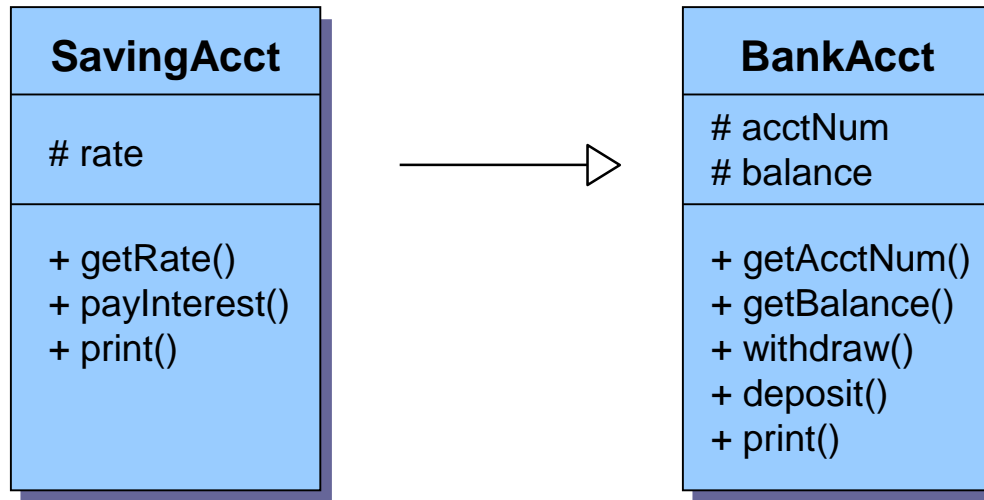
```
class SavingAcct extends BankAcct {  
    protected double rate;    // interest rate  
  
    public void payInterest() {  
        balance += balance * rate;  
    }  
  
}
```

This allows subclass of `SavingAcct` to access `rate`. If this is not intended, you may change it to “private”.

SavingAcct.java

2. Creating a Subclass (6/6)

- The subclass-superclass relationship is known as an “**is-a**” relationship, i.e. **SavingAcct is-a BankAcct**
- In the UML diagram, a solid line with a closed unfilled arrowhead is drawn from **SavingAcct** to **BankAcct**
- The symbol **#** is used to denoted protected member



2.1 Observations

- Inheritance greatly reduces the amount of redundant coding
- In `SavingAcct` class,
 - No definition of `acctNum` and `balance`
 - No definition of `withdraw()` and `deposit()`
- Improve maintainability:
 - Eg: If a method is modified in `BankAcct` class, no changes are needed in `SavingAcct` class
- The code in `BankAcct` remains untouched
 - Other programs that depend on `BankAcct` are unaffected ← very important!

2.2 Constructors in Subclass

- Unlike normal methods, constructors are NOT inherited
 - You need to define constructor(s) for the subclass

```
class SavingAcct extends BankAcct {  
    protected double rate;    // interest rate  
  
    public SavingAcct(int aNum, double bal, double rate) {  
        acctNum = aNum;  
        balance = bal;  
        this.rate = rate;  
    }  
  
    //.....payInterest() method not shown  
}
```

SavingAcct.java

2.3 The “super” Keyword

- The “**super**” keyword allows us to use the methods (including constructors) in the superclass directly
- If you make use of superclass’ constructor, it must be the **first statement** in the method body

```
class SavingAcct extends BankAcct {  
    protected double rate;    // interest rate  
    public SavingAcct(int aNum, double bal, double rate) {  
        super(aNum, bal);  
        this.rate = rate;  
    }  
  
    //.....payInterest() method not shown  
}
```

Using the constructor
in **BankAcct** class

SavingAcct.java

2.4 Using SavingAcct

TestSavingAcct.java

```
public class TestSavingAcct {  
  
    public static void main(String[] args) {  
  
        SavingAcct sa1 = new SavingAcct(2, 1000.0, 0.03);  
  
        sa1 print();  
        sa1 withdraw(50.0);  
  
        sa1 payInterest();  
        sa1 print();  
  
    }  
}
```

Inherited method from [BankAcct](#)

Method in [SavingAcct](#)

How about [print\(\)](#)?
Should it be the one in [BankAcct](#) class,
or should [SavingAcct](#) class override it?

2.5 Method Overriding (1/2)

- Sometimes we need to modify the inherited method:
 - To change/extend the functionality
 - As you already know, this is called **method overriding**
- In the `SavingAcct` class:
 - The `print()` method inherited from `BankAcct` should be modified to include the interest rate in output
- To override an inherited method:
 - Simply recode the method in the subclass using the same method header
 - Method header refers to the name and parameters type of the method (also known as **method signature**)

2.5 Method Overriding (2/2)

SavingAcct.java

```
class SavingAcct extends BankAcct {  
  
    protected double rate;    // interest rate  
  
    public double getRate() { return rate; }  
  
    public void payInterest() { ... }  
  
    public void print() {  
        System.out.println("Account Number: " + getAcctNum());  
        System.out.printf("Balance: $%.2f\n", getBalance());  
        System.out.printf("Interest: %.2f%%\n", getRate());  
    }  
}
```

- The first two lines of code in `print()` are exactly the same as `print()` of `BankAcct`
 - Can we reuse `BankAcct`'s `print()` instead of recoding?

2.6 Using “super” Again

- The `super` keyword can be used to invoke superclass' method
 - Useful when the inherited method is overridden

SavingAcct.java

```
class SavingAcct extends BankAcct {  
  
    . . .  
  
    public void print() {  
        super.print();  
        System.out.printf("Interest: %.2f%%\n", getRate());  
    }  
}
```

To use the `print()` method from `BankAcct`

3. Subclass Substitutability (1/2)

- An added advantage for inheritance is that:
 - Whenever a super class object is expected, a sub class object **is acceptable as substitution!**
 - **Caution:** the **reverse is NOT true** (Eg: A cat is an animal; but an animal may not be a cat.)
 - Hence, all existing functions that works with the super class objects will work on subclass objects with **no modification!**
- Analogy:
 - We can drive a car
 - Honda is a car (Honda is a subclass of car)
 - We can drive a Honda

3. Subclass Substitutability (2/2)

TestAcctSubclass.java

```
public class TestAcctSubclass {  
  
    public static void transfer(BankAcct fromAcct,  
                               BankAcct toAcct, double amt) {  
        fromAcct.withdraw(amt);  
        toAcct.deposit(amt);  
    };  
  
    public static void main(String[] args) {  
  
        BankAcct ba = new BankAcct(1, 234.56);  
        SavingAcct sa = new SavingAcct(2, 1000.0, 0.03);  
  
        transfer(ba, sa, 123.45);  
  
        ba.print();  
        sa.print();  
    }  
}
```

`transfer()` method can work
on the `SavingAcct` object `sa`!

4. The “Object” Class

- In Java, all classes are descendants of a predefined class called **Object**
 - **Object** class specifies some basic behaviors common to all objects
 - Any methods that works with **Object** reference will work on **object of any class**
 - Methods defined in the **Object** class are inherited in all classes
 - Two inherited **Object** methods are
 - `toString()` method
 - `equals()` method
 - However, these inherited methods usually don't work because they are not customised

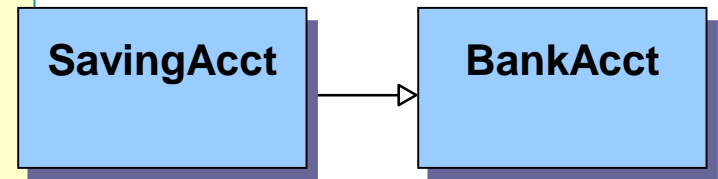
5. “is-a” versus “has-a” (1/2)

- Words of caution:
 - Do not overuse inheritance
 - Do not overuse **protected**
 - Make sure it is something inherent for future subclass
- To determine whether it is correct to inherit:
 - Use the “**is-a**” rules of thumb
 - If “B is-a A” sounds right, then ***B is a subclass of A***
 - Frequently confused with the “**has-a**” rule
 - If “B has-a A” sounds right, then ***B should have an A attribute*** (hence B depends on A)

5. "is-a" versus "has-a" (2/2)

■ UML diagrams

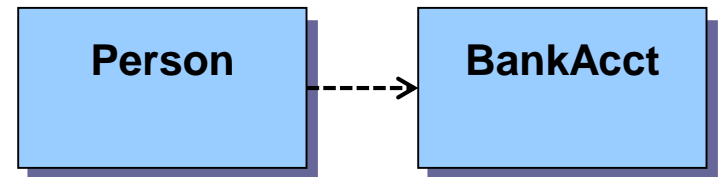
```
class BankAcct {  
    ...  
}  
  
class SavingAcct extends BankAcct {  
    ...  
}
```



Solid arrow

Inheritance: SavingAcct **IS-A** BankAcct

```
class BankAcct {  
    ...  
};  
  
class Person {  
    private BankAcct myAcct;  
};
```



Dotted arrow

Attribute: Person **HAS-A** BankAcct

6. Preventing Inheritance (“final”)

- Sometimes, we want to prevent inheritance by another class (eg: to prevent a subclass from corrupting the behaviour of its superclass)
- Use the **final** keyword
 - Eg: `final class SavingAcct` will prevent a subclass to be created from `SavingAcct`
- Sometimes, we want a class to be inheritable, but want to prevent some of its methods to be overridden by its subclass
 - Use the **final** keyword on the particular method:
`public final void payInterest() { ... }`
will prevent the subclass of `SavingAcct` from overriding `payInterest()`

7. Constraint of Inheritance in Java

- **Single inheritance**: Subclass can only have a single superclass
- **Multiple inheritance**: Subclass may have more than one superclass
- In Java, **only single inheritance is allowed**
- (Side note: Java's alternative to multiple inheritance can be achieved through the use of interfaces – to be covered later. A Java class may implement multiple interfaces.)

8. Quick Quiz #1 (1/2)

ClassA.java

```
class ClassA {
    protected int value;

    public ClassA() { }

    public ClassA(int val) { value = val; }

    public void print() {
        System.out.println("Class A: value = " + value);
    }
}
```

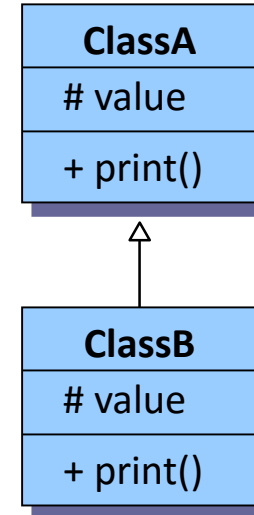
```
class ClassB extends ClassA {
    protected int value;

    public ClassB() { }

    public ClassB(int val) {
        super.value = val - 1;
        value = val;
    }

    public void print() {
        super.print();
        System.out.println("Class B: value = " + value);
    }
}
```

ClassB.java



8. Quick Quiz #1 (2/2)

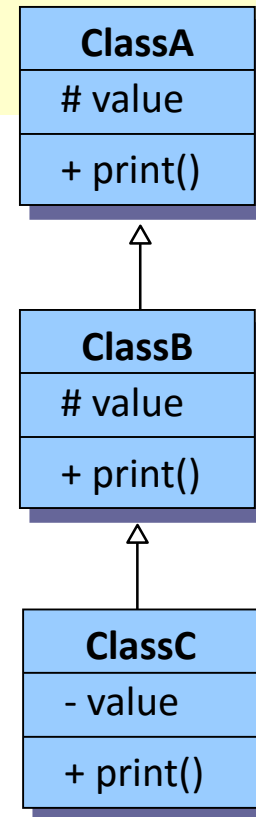
ClassC.java

```
final class ClassC extends ClassB {
    private int value;

    public ClassC() { }

    public ClassC(int val) {
        super.value = val - 1;
        value = val;
    }

    public void print() {
        super.print();
        System.out.println("Class C: value = " + value);
    }
}
```



What is the output?

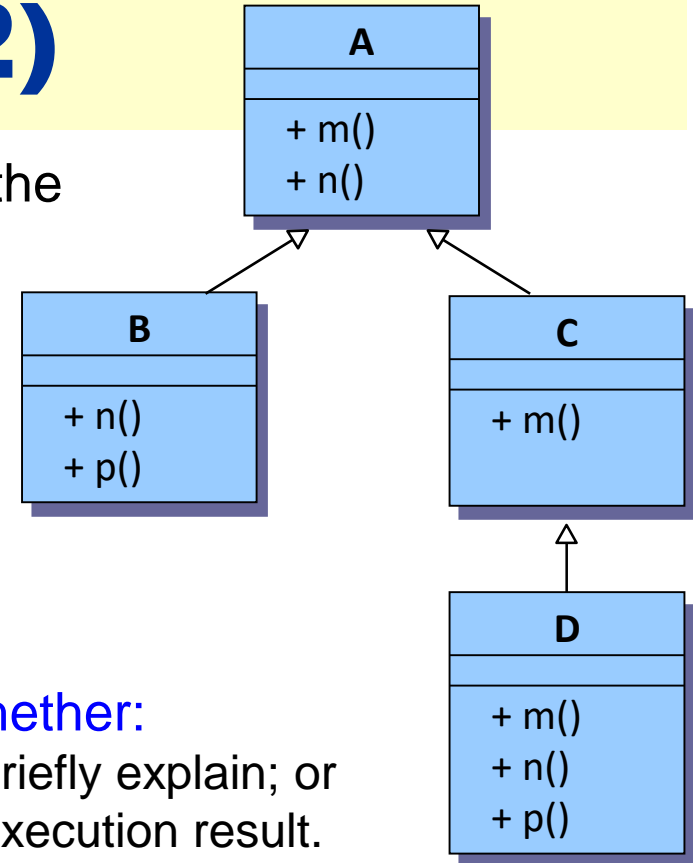
```
public class TestSubclasses {
    public static void main(String[] args) {
        ClassA objA = new ClassA(123);
        ClassB objB = new ClassB(456);
        ClassC objC = new ClassC(789);

        objA.print(); System.out.println("-----");
        objB.print(); System.out.println("-----");
        objC.print();
    }
}
```

TestSubclasses.java

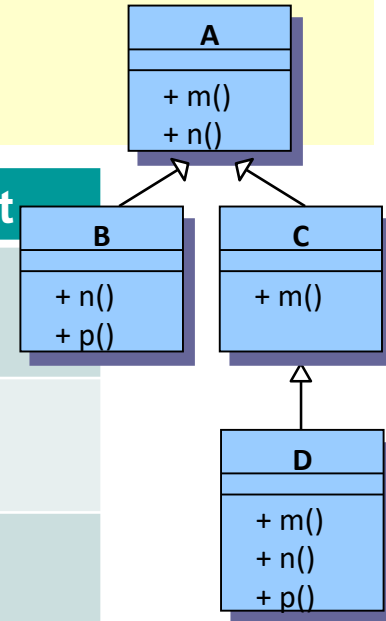
8. Quick Quiz #2 (1/2)

- Assume all methods print out message of the form <class name>,<method name>
- Eg: method m() in class A prints out “A.m”.
- If a class overrides an inherited method, the method’s name will appear in the class icon. Otherwise, the inherited method remains unchanged in the subclass.
- For each code fragment below, indicate whether:
 - The code will cause compilation error, and briefly explain; or
 - The code can compile and run. Supply the execution result.



Code fragment (example)	Compilation error? Why?	Execution result
<pre>A a = new A(); a.m();</pre>		A.m
<pre>A a = new A(); a.k();</pre>	Method k() not defined in class A	

8. Quick Quiz #2 (2/2)



Code fragment	Compilation error?	Execution result
<pre>A a = new C(); a.m();</pre>		
<pre>B b = new A(); b.n();</pre>		
<pre>A a = new B(); a.m();</pre>		
<pre>A a; C c = new D(); a = c; a.n();</pre>		
<pre>B b = new D(); b.p();</pre>		
<pre>C c = new C(); c.n();</pre>		
<pre>A a = new D(); a.p();</pre>		

Summary

- Inheritance:
 - Creating subclasses
 - Overriding methods
 - Using “super” keyword
 - The “Object” class

Practice Exercise

- Practice Exercises
 - #22: Create a subclass `CentredCircle` from a given class `Circle`
 - #23: Manage animals

End of file
