4. Java OOP

4. Inheritance and Polymorphism

Inheritance Basics (1 of 3)

- Classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes:

```
class Sub extends Sup {
    …
}
```

Inheritance Basics (2 of 3)

- 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*).
- Every class has one and only one direct superclass (single inheritance).
- Class Object is exception, it is a root class

Inheritance Basics (3 of 3)

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

Members Inheritance

- 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

Fields Inheritance

- The inherited fields can be used directly
- 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.

What will be the output?

```
class A{
   int v1 = 8;
   protected double p = -5.0;
   private String s = “1234”;
}
class B extends A{
   public void doSomething(){
          System.out.println(s);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.doSomething();
   }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected double p = -5.0;
   private String s = “1234”;
}
class B extends A{
   public void doSomething(){
          System.out.println(s);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.doSomething();
   }
}
```

Compilation error

What will be the output?

```
class A{
   int v1 = 8;
   protected double p = -5.0;
   private String s = “1234”;
}
class B extends A{
   public void doSomething(){
          System.out.println(p);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.doSomething();
   }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected double p = -5.0;
   private String s = “1234”;
}
class B extends A{
   public void doSomething(){
           System.out.println(p);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.doSomething();
   }
}
```

-5.0

What will be the output?

```
class A{
   int v1 = 8;
   protected double p = -5.0;
   private String s = “1234”;
}
class B extends A{
   public void doSomething(){
          System.out.println(v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.doSomething();
   }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected double p = -5.0;
   private String s = “1234”;
}
class B extends A{
   public void doSomething(){
           System.out.println(v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
           B obj = new B();
           obj.doSomething();
   }
}
```

8 if B and A in the same package; Compilation error otherwise

Methods Inheritance

- The inherited methods can be used directly as they are.
- You can declare new methods in the subclass that are not in the superclass.

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.printV1();
          obj.doSomething();
   }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
        System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
       System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.printV1();
          obj.doSomething();
   }
}
```

8 16

Methods Overriding and Hiding

- 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.

Constructors Call

- 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.

Exercise 4.4.1: DepoBase class

- Modify 433DepoMonthCapitalize, 432DepoBarrier, and 431SimpleDepo projects with help of ancestor DepoBase class (should contain all common elements – fields and methods)

DepoBase Class (1 of 2)

```
public class DepoBase {
    protected Date startDate;
    protected int dayLong;
    protected double sum;
    protected double interestRate;

    public DepoBase() {}

    public DepoBase(Date startDate, int dayLong, double
  sum,
                    double interestRate){
        this.startDate = startDate;
        this.dayLong = dayLong;
        this.sum = sum;
        this.interestRate = interestRate;   }
```

DepoBase Class (2 of 2)

```
// accessors

public double calculateInterest(LocalDate start, LocalDate
     maturity){
     int startYear = start.getYear();
     int maturityYear = maturity.getYear();
   . . . . . . . . . . .
     double dayCf = start.until(maturity, ChronoUnit.DAYS)
          + 1;
    double interest = sum * (interestRate / 100.0) *
          (dayCf / daysInYear);
    return interest;
}
```

DepoSimple Class

```
public class DepoSimple extends DepoBase{
    public DepoSimple(){   }
    public DepoSimple(Date startDate, int dayLong, double
            sum, double interestRate){
        super(startDate, dayLong, sum, interestRate);
    }

public double getInterest(){
    double interest = 0.0;
 . . . . . . . . . . . .
    return interest;
}
```

Exercise 4.4.1: DepoBase class

- See 441DepoBase projects for the full text

Casting Objects (1 of 3)

- *Casting* shows the use of an object of one type in place of another type, among the objects permitted by inheritance:

  Object obj = new *ClassName*();

- If, on the other hand, we write

  ClassName cn = obj;

we would get a compile-time error because obj is not known to the compiler to be a *ClassName*

Casting Objects (2 of 3)

- We can *tell* the compiler to assign a *ClassName* to obj by *explicit casting:* *ClassName* cn = (*ClassName*)obj;
- This cast inserts a runtime check that obj is assigned a *ClassName* so that the compiler can safely assume that obj is a *ClassName*
- If obj is not a *ClassName* at runtime, a ClassCastException will be thrown.

Casting Objects (3 of 3)

- You can make a logical test as to the type of a particular object using the instanceof operator:

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

- The test x instanceof C does not generate an exception if x is null. It simply returns false.

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          A obj = new B();
          obj.printV1();
          obj.doSomething();
   }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          A obj = new B();
          obj.printV1();
          obj.doSomething();
   }
}
```

Compilation error “Undefined method” on line obj.doSomething();

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
        System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new A();
          obj.printV1();
          obj.doSomething();
   }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new A();
          obj.printV1();
          obj.doSomething();
   }
}
```

Compilation error “Type mismatch” on line B obj = new A();

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = (B)new A();
          obj.printV1();
          obj.doSomething();
   }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = (B)new A();
          obj.printV1();
          obj.doSomething();
   }
}
```

}

Runtime error “ClassCastException” on line B obj = (B)new A();

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          A objA = obj;
          objA.printV1();
      }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void doSomething(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          A objA = obj;
          objA.printV1();
   }
}
```

8

Overriding Instance Methods I

- An instance method in a subclass with the same signature and return type as an instance method in the superclass *overrides* the superclass's method
- 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*.

Overriding Instance Methods II

- When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass.
- The access specifier for an overriding method can allow more, but not less, access than the overridden method (protected to public, but not to private)

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void printV1(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
          B obj = new B();
          obj.printV1();
     }
}
```

What will be the output?

```
class A{
   int v1 = 8;
   protected void printV1(){
       System.out.println(v1);
  }
}
class B extends A{
   public void printV1(){
      System.out.println(2 *
    v1);
}
```

```
Class C{
   public static void
    main(String[] args) {
           B obj = new B();
           obj.printV1();
      }
}
```

16

Hiding Static Methods (1 of 6)

```
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.");
    }
}
```

Hiding Static Methods (2 of 6)

```
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.");
    }
}
```

Hiding Static Methods (3 of 6)

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

Hiding Static Methods (4 of 6)

Output:

- The instance method in Animal
- The instance method in Cat
- The instance method in Cat

Hiding Static Methods (5 of 6)

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

Hiding Static Methods (6 of 6)

Output:

- The class method in Animal.
- The class method in Animal.
- The class method in Cat.

Polymorphism (1 of 2)

- Connecting a method call to a method body is called binding
- When binding is performed before the program is run (e.g. by the compiler), it's called early binding.
- Late binding means that the binding occurs at run time, based on the type of object
- There must be some mechanism to determine the type of the object at run time and to call the appropriate method

Polymorphism (2 of 2)

- All method binding in Java uses late binding unless the method is static or final (private methods are implicitly final)
- You can write your code to talk to the base class and know that all the derived-class cases will work correctly using the same code
- Typical example: create an array of Base class and fill it with subclasses objects. Then you can call the same method for each object from array elements

Exercise 4.4.2

- Create a deposit array of different types and calculate sum of their interest values

Type	Start	Long	Sum	Rate
Simple	08.09.2013	20	1000.00	15.0
Simple	08.09.2013	20	2500.00	18.0
Barrier	08.09.2013	40	15000.00	11.5
Barrier	08.09.2013	80	5000.00	14.0
MonthCap	08.09.2013	180	2000	16.5
MonthCap	08.09.2013	91	40000	12.1

Sum = 1763.41

Exercise: Interest Values Sum

```
Date start = new GregorianCalendar(2013,
      Calendar.SEPTEMBER, 8).getTime();
DepoBase[] depo = new DepoBase[6];
depo[0] = new DepoSimple(start, 20, 1000.0, 15.0);
depo[1] = new DepoSimple(start, 20, 2500.0, 18.0);
depo[2] = new DepoBarrier(start, 40, 15000.0, 11.5);
depo[3] = new DepoBarrier(start, 80, 5000.0, 14.0);
depo[4] = new DepoMonthCapitalize(start, 180, 2000.0, 16.5);
depo[5] = new DepoMonthCapitalize(start, 91, 40000.0, 12.1);
```

Exercise: Interest Values Sum

```
double sum = 0.0;
    for(DepoBase d: depo) sum += d.getInterest();
    sum = Math.round(sum * 100) / 100.0;
    if (sum == 1763.41) System.out.println("Test is true");
    else System.out.println("Test failed");
}
```

Exercise : Interest Values Sum

- See 442InterestSum or 442aInterestSum project for the full text

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
- Hided field in the superclass can be accessed through super keyword
- Hiding fields is not recommended as it makes code difficult to read

Subclass Constructors (1 of 2)

- The syntax for calling a superclass constructor is

  super();   or: super(parameter list);

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

Subclass Constructors (2 of 2)

- 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

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

Writing Final Methods

- You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses
- You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object
- Methods called from constructors should generally be declared final
- If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results

Final Classes

- You can declare an entire class final
- A class that is declared final cannot be subclassed
- This is particularly useful, for example, when creating an immutable class like the String class.

Manuals

- http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html