Head First Java: Chapter 7 Notes

August 27, 2019

Superclass, Subclass, and Inheritance

For efficient object-oriented programming, a more abstract *superclass* can be defined first with its own instance variables and methods, and various *subclasses* can be defined such that that it *inherits* all the instance variables and methods from the superclass. Each subclass can be further extended beyond the instanace variables and methods they inherited, even *overriding* the methods of the superclass. Note that there can be multiple levels of inheritance. A subclass can be a superclass that has subclasses of its own. It is important not to abuse inheritance by applying the IS-A and HAS-A tests:

1. IS-A Test — This test can be used to verify valid inheritance relationships. For example: class Triangle inherits from class Shape, this is valid because a triangle IS-A Shape. Other examples: Cat IS-A Feline; Surgeon IS-A Doctor.

2. HAS-A Test — Sometimes certain class relationships can seem like inheritance but it deceptively is not. For example, if class Tub extends from class Bathroom, this would NOT be a valid inheritance relationship, as a Tub is NOT a Bathroom. Rather, a Bathroom HAS-A Tub, which implies that in terms of Java code, Tub should be an instance variable of Bathroom, NOT a subclass.

The above two tests works over multiple layers of inheritance. Note that inheritance is directional; that is, a class Triangle IS-A class Shape, but a class Shape is NOT a Triangle. This directional implication means that a Triangle can do everything a Shape can do (factoring any overriding methods), but a Shape cannot necessarily (most likely not) do everything a Triangle can.

In cases where a user want to use a method of a subclass but also the same method of its superclass, this is possible:

```
public class Animal {
    void roam() {
    System.Out.print(‘‘Animal roaming’’);
    }
}

//

public class Cat extends Animal {
    void roam() {
    super.roam();
```

```
    System.Out.print(''Cat roaming'');
    }
}
```

Here, `Cat` is a subclass of `Animal`, to define it's own `roam();` method it is possible to extend the original `roam();` method from `Animal` without completely replacing it by using `super.roam();` as above.

Inheritance Access Levels

It is possible to limit class members (instance variables and methods, among other things) inheritance by flagging their access levels:

1. Public — These instance variables and methods are inherited by subclasses.
2. Private — These instance variables and methods are NOT inherited by subclasses.
3. Final — Methods flagged as final cannot be overridden.
4. Default — Related to deployment.
5. Protected — Related to deployment.

Classes themselves cannot be flagged as `private` *per se*, but if it is not flagged as `public` it can only be inherited by other classes in the same package. Additionally, a class flagged `final` cannot be inherited.

Polymorphism

Polymorphism is a mechanism that allows for great flexibility in coding with regards to types due to object inheritance. For example, given that `Dog` is a subclass of `Animal`:

```
Dog myDog = new Dog();
Animal myDog2 = new Dog();
```

Both lines of code above would be valid due to polymorphism. The implication here is that an array or an `ArrayList` can contain objects of different subclasses providing that they all have the same supers. For example:

```
Animal[] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Wolf();
animals[3] = new Hippo();
animals[4] = new Lion();
```

The above listing would be valid if `Dog, Cat, Wolf, Hippo`, and `Lion` are all subclasses of `Animal`. Looping (e.g., with a (for) loop) across the array (or ArrayList) above, even though it technically contains elements that are of different object class, would be valid. Furthermore, this has the same implication with method arguments and return types:

```
class Vet {
    public void giveShot(Animal a) {
    a.makeNoise();
    }
}

class PetOwner {
    public void start() {
    Vet v = new Vet();
    Animal d = new Dog();
    Hippo h = new Hippo();
    v.giveShot(d);
    v.giveShot(h):
    }
}
```

In the above listing, the `Vet` class has a method `giveShot();` that takes an `Animal` argument. As shown here, passing a `Dog` or a `Hippo` are both valid because they are both subclasses of `Animal` due to polymorphism.

Overriding and Overloading

It is possible for a subclass to modify a method it inherited from its superclass, this is a mechanism called *overriding*. Requirements for overriding methods:

1. Naturally, the overriding method must be named the same as the inherited method.
2. The overriding method must accept the exact same argument(s) as the inherited method it is overriding.
3. If the inherited method returns anything, the overriding method must declare the same return type or one of its subclass.
4. An overriding method cannot have less accessibility than the inherited method. For example, if the inherited method was declared as a `public` method, the overriding method cannot be declared `private`.

It is possible to declare a method with the same name as an inherited method from a superclass that does not fulfill one or more

of the above requirement. However, the resulting method will *not* have overridden the superclass method, as it actually only created a new method that coexists with the inherited superclass method, even though they would have the same name. This is refer to as *overloading*, and no polymorphism took place. There are some nuances to overloading a method:

1. As above, the name of the method in the subclass and the superclass must be the same.

2. The argument(s) of the subclass *must* be different from the inherited superclass method, otherwise the compiler will assume that the subclass method is trying to override the superclass method, and an exception might occur.

3. The return type of the subclass method do not have to be the same as the inherited superclass method.

4. The accessibility of subclass method and the superclass method do not have to be the same; they can increase or decrease in accessibility (`public` to `private` or *vice versa*).