SCOTT PANTALL

Software Developer/Tech Blogger

SEPTEMBER 12, 2017 BY SCOTTPANTALL

Four Principles of Object-Oriented Programming with Examples in Java

I enjoy object-oriented programming. I understand the concepts, but I've found that when asked to define or show an example of the basic principles my brain blanks. This especially happens in pressure situations like interviews. So this post gets to act as my memory until the four principles of object-oriented programming (encapsulation, inheritance, polymorphism and abstraction).

Encapsulation

Encapsulation is the idea that the attributes of an entity are enclosed in that entity. This gives context to attributes. This also allows the programmer to restrict access to those attributes so that those attributes are modified and/or used only in ways that the programmer intends to use them.

```
1  // This variable is not encapsulated.
2  // Therefore it's missing some context. What is it naming? I dunno.
3  String name;
4
5  // BASIC ENCAPSULATION
6  // These variables and methods are encapsulated in the Dog class, so
7  // they make more sense now. They are members of the Dog class.
8  class Dog {
9      String name;
10     int age;
```

```
11
12      void bark() {
13          System.out.println("Bark!");
14      }
15
16      void rename(String newName) {
17          name = newName;
18      }
19  }
20
21  // ACCESS MODIFIERS
22  // The members above have context, but they are accessible by any other
23  // code. To define access, use access modifiers:
24  // - default: If there is no access modifier, the attribute is available
25  //    only by classes within the same package.
26  // - public: The attribute is accessible from any other class.
27  // - protected: Same as default plus it's avaiable to subclasses
28  // - private: accessible only within the declared class.
29
30  class Dog {
31      private String name;
32      private int age;
33
34      void bark() {
35          System.out.println("Bark!");
36      }
37
38      void rename(String newName) {
39          name = newName;
40      }
41
42      public String getName() {
43          return name;
44      }
45
46      public void setAge(int newAge) {
47          if(newAge > 0)
48              age = newAge;
49      }
50
51      public int getAge() {
52          return age;
```

```
    }
}
```

encapsulation.java hosted with ♥ by GitHub view raw

Inheritance

Inheritance is the idea that an entity can inherit attributes from another entity. It allows the programmer to create similar entities without needing to redefine similar attributes over and over.

```java
// If we want to define dogs and cats and birds and we know all these
// animals have names and ages we should create a superclass class.

// SUPERCLASS
class Animal {
    private String name;
    private int age;

    public void identify() {
        System.out.println("I am an animal!");
    }

    public void rename(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }

    public void setAge(int newAge) {
        if(newAge > 0)
            age = newAge;
    }

    public int getAge() {
        return age;
    }
```

```
29  }
30
31  // SUBCLASSES
32  class Dog extends Animal {
33      public void bark() {
34          System.out.println("Bark!");
35      }
36  }
37
38  class Cat extends Animal {
39      public void meow() {
40          System.out.println("Meow!");
41      }
42  }
43
44  class Bird extends Animal {
45      public void chirp() {
46          System.out.println("Chirp!");
47      }
48  }
```

inheritance.java hosted with ♥ by GitHub view raw

This means every Dog, Cat and Bird (subclasses) will have a name and age attributes as well as an Identify method because they are Animals (superclass).

Polymorphism

Polymorphism means “having many forms” which honestly doesn’t really help too much as a definition. I like to call it a way to have the same method, only different. There are two ways to do this:

- Overloading: The method name stays the same, but the parameters, the return type and the number of parameters can all change.

```
1  // OVERLOADING
2  // The Animal class shows 3 Constructor methods that are overloaded.
3  // This means we can create a new Animal 3 different ways.
4  class Animal {
```

```
    private String name;
    private int age;

    Animal() {
        name = "Fred";
        age = 12;
    }

    Animal(String nm) {
        name = nm;
        age = 5;
    }

    Animal(String nm, int newAge) {
        name = nm;
        age = newAge;
    }
    .
    .
    .
}
```

overloading.java hosted with ♥ by GitHub view raw

- Overriding: This is when a subclass method has the same name, parameters and return type as a method in a superclass but has a different implementation.

```
// SUPERCLASS
class Animal {
    .
    .
    .
    public void identify() {
        System.out.println("I am an animal!");
    }

}

// SUBCLASSES
class Dog extends Animal {
    public void bark() {
        System.out.println("Bark!");
    }

```

```
18      public void identify() {
19          System.out.println("I am a dog!");
20      }
21  }
22
23  class Cat extends Animal {
24      public void meow() {
25          System.out.println("Meow!");
26      }
27
28      public void identify() {
29          System.out.println("I am a cat!");
30      }
31  }
32
33  class Bird extends Animal {
34      public void chirp() {
35          System.out.println("Chirp!");
36      }
37
38      public void identify() {
39          System.out.println("I am a bird!");
40      }
41  }
```

overriding.java hosted with ♥ by GitHub view raw

A few rules about method overriding in Java...

- Subclass methods should have the same return type and arguments
- The access level of the subclass method cannot be more restrictive than the superclass method.
- A method declared final or static cannot be overridden.
- If a method cannot be inherited, it cannot be overridden.

Abstraction

Abstraction is the process of hiding all but the relevant information about a thing to make things less complex and more efficient for the user. For example, we don't need to know how a clock works in order to use it to tell time. Abstraction lets you focus on what the thing does instead of how it does it.

```
// BASIC ABSTRACTION
// We can use System.out.println("Hello") to write a string to the
// console without worrying about how the println method works.

// ABSTRACT CLASSES
// An abstract class only has method declarations. it is intended to be
// a superclass for other classes. It doesn't define how methods are
// implemented, only that they are implemented. Abstract classes cannot
// be instantiated and subclasses MUST implement abstract methods.

abstract class FlyingAnimal {
    public abstract void Fly();
}

class Bird extends FlyingAnimal {
    protected String name;
    protected int age;

    Bird(String nm, int newAge) {
        name = nm;
        age = newAge;
    }

    @Override
    public void Fly() {
        System.out.println("Flaps wings majestically.");
    }
}

// INTERFACES
// Classes can only inherit from one superclass, but they can implement
// multiple interfaces. This expands our ability to make good use of
// abstraction. Classes that implement interfaces MUST implement the
// methods in the interface.

// Regular class
class Animal {
```

```
38      private String name;
39      private int age;
40
41      public void identify() {
42          System.out.println("I am an animal!");
43      }
44
45      public void rename(String newName) {
46          name = newName;
47      }
48
49      public String getName() {
50          return name;
51      }
52
53      public void setAge(int newAge) {
54          if(newAge > 0)
55              age = newAge;
56      }
57
58      public int getAge() {
59          return age;
60      }
61  }
62
63  // Interface for things that fly. We don't care how they fly, just that
64  // they can fly.
65  public interface ICanFly {
66      void Fly();
67  }
68
69  // Interface for things that swim. We don't care how they swim, just
70  // that they can swim.
71  public interface ICanSwim {
72      void Swim();
73  }
74
75  // A Duck is an Animal that can fly and swim.
76  class Duck extends Animal implements ICanFly, ICanSwim {
77      public void Quack() {
78          System.out.println("QUACK!");
79      }
```

```
80
81      @Override
82      public void Identify() {
83          System.out.println("I am a duck!");
84      }
85
86      @Override
87      public void Fly() {
88          System.out.println("Flaps wings majestically.");
89      }
90
91      @Override
92      public void Swim() {
93          System.out.println("Kicks feet.");
94      }
95  }
96
97  // A Fish is an Animal that can swim. Notice the implementation of the
98  // Swim method is different for a fish than a duck.
99  class Fish extends Animal implements ICanSwim {
100     @Override
101     public void Identify() {
102         System.out.println("I am a fish!");
103     }
104
105     @Override
106     public void Swim() {
107         System.out.println("Wiggles fish-body");
108     }
109 }
110
111 // An AirPlane is not an animal, but it can still fly.
112 class AirPlane implements ICanFly {
113     protected String name;
114     protected int mileage;
115
116     @Override
117     public void Fly() {
118         System.out.println("Turns propeller");
119     }
120 }
```

abstraction.java hosted with ♥ by GitHub view raw

As much as I tried to keep thins post brief, I failed. If I got something wrong, or if something doesn't make sense please reach out and let me know.

Here are a few of the resources I used:

- https://docs.oracle.com/javase/tutorial/java/concepts/index.html
- https://www.sololearn.com/

TECHNOLOGY