Multiple Inheritance in Java and Composition vs Inheritance

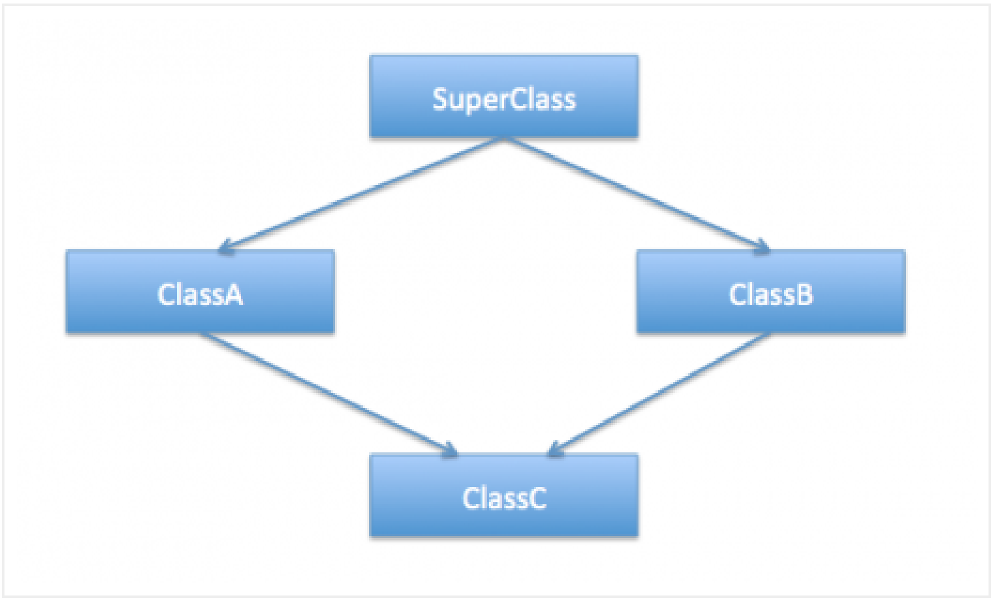
Sometime back I wrote few posts about inheritance, interface and composition in java. In this post, we will look into multiple inheritance and then learn about benefits of composition over inheritance.

Multiple Inheritance in Java

Multiple inheritance is the capability of creating a single class with multiple superclasses. Unlike some other popular object oriented programming languages like C++, **java doesn't provide support for multiple inheritance in classes**. Java doesn't support multiple inheritance in classes because it can lead to **diamond problem** and rather than providing some complex way to solve it, there are better ways through which we can achieve the same result as multiple inheritance.

Diamond Problem

To understand diamond problem easily, let's assume that multiple inheritance was supported in java. In that case, we could have a class hierarchy like below image.



Let's say SuperClass is an abstract class declaring some method and ClassA, ClassB are concrete classes.

SuperClass.java

```java
package com.journaldev.inheritance;

```

```
public abstract class SuperClass {

    public abstract void doSomething();
}
```

ClassA.java

```
package com.journaldev.inheritance;

public class ClassA extends SuperClass{

    @Override
    public void doSomething(){
        System.out.println("doSomething implementation of A");
    }

    //ClassA own method
    public void methodA(){

    }
}
```

ClassB.java

```
package com.journaldev.inheritance;

public class ClassB extends SuperClass{

    @Override
    public void doSomething(){
        System.out.println("doSomething implementation of B");
    }

    //ClassB specific method
    public void methodB(){

    }
}
```

Now let's say ClassC implementation is something like below and it's extending both ClassA and ClassB.

ClassC.java

```
package com.journaldev.inheritance;

public class ClassC extends ClassA, ClassB{

    public void test(){
        //calling super class method
        doSomething();
    }

}
```

Notice that `test()` method is making a call to superclass `doSomething()` method, this leads to the ambiguity as compiler doesn't know which superclass method to execute and because of the diamond shaped class diagram, it's referred as Diamond Problem and this is the main reason java doesn't support multiple inheritance in classes.

Notice that the above problem with multiple class inheritance can also come with only three classes where all of them has at least one common method.

Multiple Inheritance in Interfaces

You might have noticed that I am always saying that multiple inheritance is not supported in classes but it's supported in interfaces and a single interface can extend multiple interfaces, below is a simple example.

InterfaceA.java

```java
package com.journaldev.inheritance;

public interface InterfaceA {

    public void doSomething();
}
```

InterfaceB.java

```java
package com.journaldev.inheritance;

public interface InterfaceB {

    public void doSomething();
}
```

Notice that both the interfaces are declaring same method, now we can have an interface extending both these interfaces like below.

InterfaceC.java

```java
package com.journaldev.inheritance;

public interface InterfaceC extends InterfaceA, InterfaceB {

    //same method is declared in InterfaceA and InterfaceB both
    public void doSomething();

}
```

This is perfectly fine because the interfaces are only declaring the methods and the actual implementation will be done by concrete classes implementing the interfaces, so there is no possibility of any kind of ambiguity in multiple inheritance in interface.

Thats why a java class can implement multiple inheritance, something like below example.

InterfacesImpl.java

```java
package com.journaldev.inheritance;

public class InterfacesImpl implements InterfaceA, InterfaceB, InterfaceC {

    @Override
    public void doSomething() {
        System.out.println("doSomething implementation of concrete class");
    }

    public static void main(String[] args) {
        InterfaceA objA = new InterfacesImpl();
        InterfaceB objB = new InterfacesImpl();
        InterfaceC objC = new InterfacesImpl();

        //all the method calls below are going to same concrete implementatio
        objA.doSomething();
        objB.doSomething();
        objC.doSomething();
    }

}
```

Did you noticed that every time I am overriding any superclass method or implementing any interface method, I am using @Override annotation, it's one of the three built-in **java annotations** and we should always use override annotation when overriding any method.

Composition for the rescue

So what to do if we want to utilize `ClassA` function *methodA()* and `ClassB` function *methodB()* in `ClassC`, the solution lies in using **composition**, here is a refactored version of ClassC that is using composition to utilize both classes methods and also using **doSomething()** method from one of the objects.

ClassC.java

```java
package com.journaldev.inheritance;

public class ClassC{

    ClassA objA = new ClassA();
    ClassB objB = new ClassB();

    public void test(){
        objA.doSomething();
    }

    public void methodA(){
        objA.methodA();
    }

    public void methodB(){
        objB.methodB();
    }
}
```

Composition vs Inheritance

One of the best practices of java programming is to "favor composition over inheritance", we will look into some of the aspects favoring this approach.

1. Suppose we have a superclass and subclass as follows:

ClassC.java

```java
package com.journaldev.inheritance;

public class ClassC{

    public void methodC(){
    }
}
```

ClassD.java

```java
package com.journaldev.inheritance;

public class ClassD extends ClassC{

    public int test(){
        return 0;
    }
}
```

The above code compiles and works fine but what if ClassC implementation is changed like below:

ClassC.java

```java
package com.journaldev.inheritance;

public class ClassC{

    public void methodC(){
    }

    public void test(){
    }
}
```

Notice that *test()* method already exists in the subclass but the return type is different, now the ClassD won't compile and if you are using any IDE, it will suggest you to change the return type in either superclass or subclass.

Now imagine the situation where we have multiple level of class inheritance and superclass is not controlled by us, we will have no choice but to change our subclass method signature or it's name to remove the compilation error, also we will have to make change in all the places where our subclass method was getting invoked, so inheritance makes our code fragile.

The above problem will never occur with composition and that makes it more favorable over inheritance.

2. Another problem with inheritance is that we are exposing all the superclass methods to the client and if our superclass is not properly designed and there are security holes, then even though we take complete care in implementing our class, we get affected by the poor implementation of superclass.
Composition helps us in providing controlled access to the superclass methods whereas inheritance doesn't provide any control of the superclass methods, this is also one of the major

advantage of composition over inheritance.

3. Another benefit with composition is that it provides flexibility in invocation of methods. Our above implementation of `ClassC` is not optimal and provides compile time binding with the method that will be invoked, with minimal change we can make the method invocation flexible and make it dynamic.

   ClassC.java
   ```java
   package com.journaldev.inheritance;

   public class ClassC{

       SuperClass obj = null;

       public ClassC(SuperClass o){
           this.obj = o;
       }
       public void test(){
           obj.doSomething();
       }

       public static void main(String args[]){
           ClassC obj1 = new ClassC(new ClassA());
           ClassC obj2 = new ClassC(new ClassB());

           obj1.test();
           obj2.test();
       }
   }
   ```

   Output of above program is:

   ```
   doSomething implementation of A
   doSomething implementation of B
   ```

   This flexibility in method invocation is not available in inheritance and boosts the best practice to favor composition over inheritance.

4. Unit testing is easy in composition because we know what all methods we are using from superclass and we can mock it up for testing whereas in inheritance we depend heavily on superclass and don't know what all methods of superclass will be used, so we need to test all the methods of superclass, that is an extra work and we need to do it unnecessarily because of inheritance.

Ideally we should use inheritance only when the "**is-a**" relationship holds true for superclass and subclass in all the cases, else we should go ahead with composition.