Multiple Inheritance in Java

APRIL 2, 2018 BY PANKAJ — 16 COMMENTS

Today we will look into Multiple Inheritance in Java. Sometime back I wrote few posts about **inheritance**, **interface** and **composition** in java. In this post, we will look into java multiple inheritance and then compare composition and inheritance.

Table of Contents [hide]

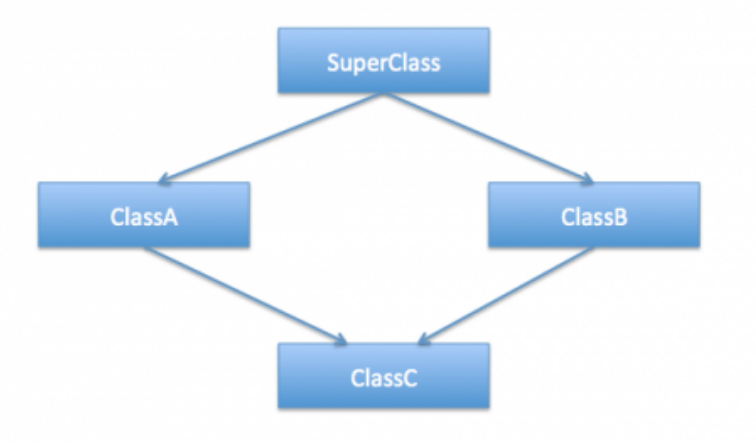
Multiple Inheritance in Java

Multiple inheritance in java 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 in Java

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

```
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. Because of the diamond shaped class diagram, it's referred as Diamond Problem in java. Diamond problem in java 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 Java Interfaces

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

`InterfaceA.java`

```
package com.journaldev.inheritance;

public interface InterfaceA {

	public void doSomething();
}
```

`InterfaceB.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

```
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 java interfaces.

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

InterfacesImpl.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
```

```
    implementation
                    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. Override annotation is 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`

```
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

```
package com.journaldev.inheritance;

public class ClassC{

        public void methodC(){
        }
}
```

ClassD.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

```
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`

```
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.

That's all for multiple inheritance in java and a brief look on composition.

« PREVIOUS
Exception Handling in Java

NEXT »
Java Reflection Example Tutorial

About Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: JAVA

Comments

Paresh Sojitra says
JUNE 9, 2018 AT 1:58 AM
Hi, We can change the access specifier as private test() method of ClassC and we again declare the same method as public in ClassD, It Will compile without error. It's doesn't matter return type as well.

```
public class ClassC{
public void methodC()
{
}
private void test()
{
}
}
public class ClassD extends ClassC {
public int test()
{
return 0;
}
}
```

Reply

subrat says
OCTOBER 10, 2017 AT 7:09 AM
sir u do not discuss abstract class before.if u take example like this how student will understand
Reply

T sinha says
AUGUST 30, 2017 AT 4:07 AM
Can you please explain the example given in point 3, public ClassC(SuperClass o){
this.obj = o;

}

.

.

.

ClassC obj1 = new ClassC(new ClassA());

ClassC obj2 = new ClassC(new ClassB());

How can this be achived?

Reply

Yashvir Singh says

JUNE 22, 2017 AT 12:01 AM

You are awesome Pankaj.

Reply

Ravi Verma says

MAY 6, 2017 AT 3:06 AM

You are awesome Pankaj.

Reply

Mayank says

DECEMBER 27, 2016 AT 4:28 AM

There are many advantages of using composition, couple of them are :

You will have full control of your implementations. i.e., you can expose only the methods you intend to expose.

any changes in the super class can be shielded by modifying only in your class. Any clients classes which uses your classes, need not make modifications.

Allows you to control when you want to load the super class (lazy loading)

Reply

Yunus Atheist says

MARCH 21, 2016 AT 8:28 AM

composition has its limitations. I believe composition is an option as long as the classes and their methods we want to consume are “public”. In case we want to access “protected” members you have to fall back on either Interfaces or Classes. Not denying – composition is a useful means and improves our code refactoring skills.

Reply

safdar says

FEBRUARY 12, 2016 AT 2:08 AM

Suppose we have a superclass and subclass as follows:

ClassC.java

```
1
2
3
4
5
6
7
package com.journaldev.inheritance;
public class ClassC{
public void methodC(){
}
}
```

ClassD.java

```
1
2
3
4
5
6
7
8
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

```
1
2
3
4
5
6
7
8
```

```
9
10
package com.journaldev.inheritance;
public class ClassC{
public void methodC(){
}
public void test(){
}
}
```

Suppose we have a superclass and subclass as follows:

ClassC.java

```
1
2
3
4
5
6
7
package com.journaldev.inheritance;
public class ClassC{
public void methodC(){
}
}
```

ClassD.java

```
1
2
3
4
5
6
7
8
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

```
1
2
3
4
5
```

```
6
7
8
9
10
package com.journaldev.inheritance;
public class ClassC{
public void methodC(){
}
public void test(){
}
}
```

I think this not the problem after Jdk 1.5 where we can have different return types for both overriding and overidden methods in the parent and subclass.

Reply

Sorrowfull Blinger says

AUGUST 28, 2014 AT 4:29 AM

Can u give examples/Complete class defintions to proove your 2nd & 3rd point of Composition better than inheritance.??

Reply

Suman says

JUNE 25, 2014 AT 1:33 PM

How the multiple inheritance is possible in C , C++ but not in java
can u explain the context…

Reply

Elumalai says

APRIL 30, 2014 AT 4:13 AM

How the multiple inheritance is possible in C , C++ but not in java
can u explain the context…

Reply

Asif Mushtaq says

OCTOBER 13, 2015 AT 10:45 AM

class A{
void method1(){}
};
class B{
void method2(){}
};
class C: public A, public B{
}
//////////////////////////
Now C have both methods .. method1 and method 2.. that's multiple inheritance in C++, but java not allowed it.
Reply

Madhusmita Nayak says
FEBRUARY 6, 2014 AT 12:26 PM
What is association in Java?
Reply

subbareddy says
JANUARY 7, 2014 AT 12:07 PM
good explanation……..
Reply

Super Hubo says
AUGUST 21, 2013 AT 2:57 PM
Bit confused. "favor composition over interfaces" or "favor composition over inheritance"?.
Reply

Pankaj says
AUGUST 21, 2013 AT 6:35 PM
Thanks for catching the typo error, corrected it.
Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

CORE JAVA TUTORIAL

Java 10 Tutorials Java 9 Tutorials Java 8 Tutorials Java 7 Tutorials Core Java Basics OOPS Concepts Data Types and Operators String Manipulation

Java Arrays Annotation and Enum
Java Collections Java IO Operations
Java Exception Handling
MultiThreading and Concurrency
Regular Expressions Advanced Java Concepts

RECOMMENDED TUTORIALS

Java Tutorials

- Java IO
- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial