Why is there no multiple inheritance in Java, but implementing multiple interfaces is allowed?

Asked 11 years, 6 months ago Active 1 month ago Viewed 166k times

175

Java doesn't allow multiple inheritance, but it allows implementing multiple interfaces. Why?

java oop inheritance interface language-design

85

Share Edit Follow

edited Sep 21 '16 at 14:28

Ravindra babu
43.8k ● 8 ● 218 ● 198

asked Mar 25 '10 at 12:40

abson
8,672 ● 15 ● 46 ● 67

1 I edited the question title to make it more descriptive. – Bozho Mar 25 '10 at 12:45

4 Interestingly, in the JDK 8, there will be extension methods which will allow the definition of implementation of interface methods. Rules are been defined to govern multiple inheritance of behavior, but not of state (which I understand is more problematic. – Edwin Dalorzo Jul 31 '12 at 12:48

1 Before you guys waste time on answers which only tell you "How java acheives multiple inheritance "... I suggest you to go down to @Prabal Srivastava answer that logically gives you what must be happening internally, to not allow classes this right... and only allow interfaces the right to allow multiple inheritance. – Yo Apps Nov 27 '19 at 15:09

20 Answers

Active Oldest Votes

251

✓

Because interfaces specify only *what* the class is doing, not *how* it is doing it.

The problem with multiple inheritance is that two classes may define *different ways* of doing the same thing, and the subclass can't choose which one to pick.

Share Edit Follow

answered Mar 25 '10 at 12:43

Bozho
561k ● 137 ● 1032 ● 1125

8 I used to do C++ and ran into that exact same issue quite a few times. I recently read about Scala having "traits" that to me seem like something in between the "C++" way and the "Java" way of doing things. – Niels Basjes Jul 29 '10 at 21:04

5 This way, you are avoiding the "diamond problem": en.wikipedia.org/wiki/Diamond_problem#The_diamond_problem – Nick Louloudakis Feb 13 '14 at 15:32

6 Since Java 8 you can define two identical default methods, one in each interface. If you will implement both interfaces in your class, you have to override this method in the class itself, see: docs.oracle.com/javase/tutorial/java/IandI/… – bobbel May 14 '14 at 12:29

9 This answer is not accurate. The problem is not specifying the *how* and Java 8 is there to prove. In Java 8, two *super interfaces* can declare the same method with different implementations, but that's not a problem because interface methods are *virtual*, so you can just overwrite them and the problem is solved. The real problem is about **ambiguity** in the **attributes**, because you can't solve this ambiguity overwriting the attribute (attributes are not virtual). – Alex Oliveira Jul 13 '16 at 19:51

1 What do you mean by "attributes"? – Bozho Jul 22 '16 at 12:20

106

One of my college instructors explained it to me this way:

> Suppose I have one class, which is a Toaster, and another class, which is NuclearBomb. They both might have a "darkness" setting. They both have an on() method. (One has an off(), the other doesn't.) If I want to create a class that's a subclass of both of these...as you can see, this is a problem that could really blow up in my face here.

So one of the main issues is that if you have two parent classes, they might have different implementations of the same feature — or possibly two different features with the same name, as in my instructor's example. Then you have to deal with deciding which one your subclass is going to use. There are ways of handling this, certainly — C++ does so — but the designers of Java felt that this would make things too complicated.

With an interface, though, you're describing something the class is capable of doing, rather than borrowing another class's method of doing something. Multiple interfaces are much less likely to cause tricky conflicts that need to be resolved than are multiple parent classes.

Share Edit Follow

answered Mar 25 '10 at 13:04
Syntactic
9,773 ● 2 ● 22 ● 25

5 Thanks, NomeN. The source of the quote was a PhD student named Brendan Burns, who at the time was also the maintainer of the open-source Quake 2 source repository. Go figure. – Syntactic Mar 25 '10 at 13:22

4 The problem with this analogy is that if you are creating a subclass of a nuclear bomb and a

toaster, the "nuclear toaster" would reasonably blow up when used. – 101100 Mar 27 '13 at 17:55

26 If someone decides to mix a nuclear bomb and a toaster, he deserves that the bomb blows up in his face. The quote is *fallacious reasoning* – masoud May 19 '13 at 12:15

1 The same programming language does allow multiple "interface" inheritance, so this "justification" does not apply. – curiousguy May 19 '13 at 16:13

1 Lol only looking at this now, but big up vote vor @masoud. Every computer science student should be asking this question – Tomvkgames Dec 11 '20 at 2:45

24

Because inheritance is overused even when you *can't* say "hey, that method looks useful, I'll extend that class as well".

```
public class MyGodClass extends AppDomainObject, HttpServlet, MouseAdapter,
                 AbstractTableModel, AbstractListModel, AbstractList, AbstractMap,
...
```

Share Edit Follow

answered Mar 25 '10 at 12:50
Michael Borgwardt
330k ●75 ●462 ●701

Can you explain why you say inheritance is overused? Creating a god class is exactly what I want to do! I find so many people that work around single inheritance by creating "Tools" classes that have static methods. – Duncan Calvert Aug 12 '14 at 1:32

10 @DuncanCalvert: No, you do not want to do that, not if that code will ever need maintenance. Lots of static methods misses the point of OO, but excessive multiple inheritance is *much* worse because you completely lose track of which code is used where, as well as what a class conceptually is. Both are trying to solve the problem of "how can I use this code where I need it", but that is a simple short-term problem. The much harder long-term problem solved by proper OO design is "how do I change this code without having the program break in 20 different places in unforeseeable ways? – Michael Borgwardt Aug 12 '14 at 13:03

3 @DuncanCalvert: and you solve that by having classes with high cohesion and low coupling, which means they contain pieces of data and code that interact intensively with each other, but interact with the rest of the program only through a small, simple public API. Then you can think about them in terms of that API instead of the internal details, which is important because people can only keep a limited amount of details in mind at the same time. – Michael Borgwardt Aug 12 '14 at 13:06

@MichaelBorgwardt your reasoning is flawed. – mmm Apr 23 at 9:57

@mmm can you point out how? – Michael Borgwardt Apr 23 at 19:42

The answer of this question is lies in the internal working of java

21

compiler(constructor chaining). If we see the internal working of java compiler:

```
public class Bank {
  public void printBankBalance(){
    System.out.println("10k");
  }
}
class SBI extends Bank{
 public void printBankBalance(){
    System.out.println("20k");
  }
}
```

After compiling this look like:

```
public class Bank {
  public Bank(){
   super();
  }
  public void printBankBalance(){
    System.out.println("10k");
  }
}
class SBI extends Bank {
 SBI(){
   super();
 }
 public void printBankBalance(){
    System.out.println("20k");
  }
}
```

when we extends class and create an object of it, one constructor chain will run till `Object` class.

Above code will run fine. but if we have another class called `Car` which extends `Bank` and one *hybrid*(multiple inheritance) class called `SBICar`:

```
class Car extends Bank {
  Car() {
    super();
  }
  public void run(){
    System.out.println("99Km/h");
  }
}
class SBICar extends Bank, Car {
  SBICar() {
    super(); //NOTE: compile time ambiguity.
  }
  public void run() {
    System.out.println("99Km/h");
  }
  public void printBankBalance(){
    System.out.println("20k");
    }
  }
```

In this case(SBICar) will fail to create constructor chain(**compile time ambiguity**).

For interfaces this is allowed because we cannot create an object of it.

For new concept of `default` and `static` method kindly refer default in interface.

Hope this will solve your query. Thanks.

Share Edit Follow edited Jan 3 '20 at 11:56 answered Mar 25 '19 at 3:41
Prabal Srivastava
764 ● 5 ● 15

8

You can find accurate answer for this query in oracle documentation page about multiple inheritance

1. ***Multiple inheritance of state:*** Ability to inherit fields from multiple classes

   > One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of multiple inheritance of state, which is the ability to inherit fields from multiple classes

   If multiple inheritance is allowed and When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. It will cause two issues.

   1. What if methods or constructors from different super classes instantiate the same field?
   2. Which method or constructor will take precedence?

2. ***Multiple inheritance of implementation:*** Ability to inherit method definitions from multiple classes

   Problems with this approach: *name conflic*ts and *ambiguity*. If a subclass and superclass contain same method name (and signature), compiler can't determine which version to invoke.

   But java supports this type of multiple inheritance with default methods, which have been introduced since Java 8 release. The Java compiler provides some rules to determine which default method a particular class uses.

   Refer to below SE post for more details on resolving diamond problem:

   What are the differences between abstract classes and interfaces in Java 8?

3. ***Multiple inheritance of type:*** Ability of a class to implement more than one interface.

   Since interface does not contain mutable fields, you do not have to worry about problems that result from multiple inheritance of state here.

Share Edit Follow

edited May 23 '17 at 10:31
Community Bot
1 ● 1

answered Sep 17 '16 at 17:12
Ravindra babu
43.8k ● 8 ● 218 ● 198

6

Implementing multiple interfaces is very useful and doesn't cause much problems to language implementers nor programmers. So it is allowed. Multiple inheritance while also useful, can cause serious problems to users (dreaded diamond of death). And most things you do with multiple inheritance can be also done by composition or using inner classes. So multiple inheritance is forbidden as bringing more problems than gains.

Share Edit Follow

answered Mar 25 '10 at 13:00
Tadeusz Kopec
12k ● 6 ● 51 ● 79

What is the problem with the "diamond of death"? – curiousguy May 20 '13 at 23:33

2 @curiousguy Containing more than one subobject of same base class, ambiguity (override from which base class use), complicated rules of resolving such ambiguity. – Tadeusz Kopec May 22 '13 at 7:54

@curiousguy: If a framework provides that casting an object reference to a base-type reference will be identity preserving, then every object instance must have exactly one implementation of any base-class method. If `ToyotaCar` and `HybridCar` both derived from `Car` and overrode `Car.Drive`, and if `PriusCar` inherited both *but didn't override* *`Drive`*, the system would have no way of identifying what the virtual `Car.Drive` should do. Interfaces avoid this problem by avoiding the italicized condition above. – supercat Jun 16 '13 at 19:36

1 @supercat " the system would have no way of identifying what the virtual Car.Drive should do." <-- Or it could just give a compile error and make you choose one explicitly, like C++ does. – Chris Middleton Aug 13 '15 at 7:03

1 @ChrisMiddleton: A method `void UseCar(Car &foo)`; cannot be expected to include disambiguation between `ToyotaCar::Drive` and `HybridCar::Drive` (since it should often neither know nor care that those other types even *exist*). A language could, as C++ does, require that code with `ToyotaCar &myCar` wishing to pass it to `UseCar` must first cast to either `HybridCar` or `ToyotaCar`, but since ((Car)(HybridCar)myCar).Drive` and `((Car)(ToyotaCar)myCar).Drive` would do different things, that would imply that the upcasts were not identity-preserving. – supercat Aug 13 '15 at 15:57

6

Java supports multiple inheritance through interfaces only. A class can implement any number of interfaces but can extend only one class.

Multiple inheritance is not supported because it leads to deadly diamond problem. However, it can be solved but it leads to complex system so multiple inheritance has been dropped by Java founders.

In a white paper titled “Java: an Overview” by James Gosling in February 1995([link - page 2]) gives an idea on why multiple inheritance is not supported in Java.

According to Gosling:

> "JAVA omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance, and extensive automatic coercions."

Share Edit Follow

edited Apr 27 at 16:46
Kyroath
71 ● 1 ● 5

answered Sep 28 '16 at 7:33
Praveen Kumar
498 ● 6 ● 15

the link is not accessible. Kindly check and update it. – MashukKhan Jun 22 '20 at 5:42

4

It is said that objects state is referred with respect to the fields in it and it would become ambiguous if too many classes were inherited. Here is the link

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

Share Edit Follow

answered Feb 13 '14 at 15:29
Karthik GVD
71 ● 1 ● 9

3

For the same reason C# doesn't allow multiple inheritence but allows you to implement multiple interfaces.

The lesson learned from C++ w/ multiple inheritence was that it lead to more issues than it was worth.

An interface is a contract of things your class has to implement. You don't gain any functionality from the interface. Inheritence allows you to inherit the functionality of a parent class (and in multiple-inheritence, that can get extremely confusing).

Allowing multiple interfaces allows you to use Design Patterns (like Adapter) to solve the same types of issues you can solve using multiple inheritence, but in a much more reliable and predictable manner.

Share Edit Follow edited Mar 25 '10 at 12:47 answered Mar 25 '10 at 12:42

Justin Niessner
232k ● 36 ● 397 ● 525

10 C# does not have multiple inheritance precisely because Java does not allow it. It was designed much later than Java. The main problem with multiple inheritance I think was the way people were taught to use it left and right. The concept that delegation in most cases is a much better alternative just was not there in the early and mid-nineties. Hence I remember examples, in textbooks, when Car is a Wheel and a Door and a Windshield, vs. Car contains Wheels, Doors and Windshield. So the single inheritance in Java was a knee jerk reaction to that reality. – Alexander Pogrebnyak Mar 25 '10 at 12:55

2 @AlexanderPogrebnyak: Pick two of the following three: (1) Allow identity-preserving casts from a subtype reference to a supertype reference; (2) Allow a class to add virtual public members without recompiling derived classes; (3) Allow a class to implicitly inherit virtual members from multiple base classes without having to explicitly specify them. I don't believe it's possible for any language to manage all three of the above. Java opted for #1 and #2, and C# followed suit. I believe C++ adopts #3 only. Personally, I think #1 and #2 are more useful than #3, but others may differ. – supercat Jun 16 '13 at 19:45

@supercat "I don't believe it's possible for any language to manage all three of the above" – if the offsets of data members are determined at runtime, rather than compilation time (as they are in Objective-C's "non-fragile ABI"), and or a per-class basis (i.e. each concrete class has its own member offset table), then I think all 3 goals can be achieved. – The Paramagnetic Croissant Sep 27 '15 at 18:46

@TheParamagneticCroissant: The major semantic problem is #1. If `D1` and `D2` both inherit from `B`, and each overrides a function `f`, and if `obj` is an instance of a type `S` which inherits from both `D1` and `D2` but does not override `f`, then casting a reference to `S` to `D1` should yield something whose `f` uses the `D1` override, and casting to `B` shouldn't change that. Likewise casting a reference `S` to `D2` should yield a something whose `f` uses the `D2` override, and casting to `B` shouldn't change that. If a language didn't need to allow virtual members to be added... – supercat Sep 27 '15 at 19:11

1 @TheParamagneticCroissant: It could, and my list of choices was somewhat over-simplified to fit in a comment, but deferring to run-time makes it impossible for the author of D1, D2, or S to know what changes they can make without breaking consumers of their class. – supercat Sep 27 '15 at 20:20

3

Java does not support multiple inheritance because of two reasons:

1. In java, every class is a child of `Object` class. When it inherits from more than one super class, sub class gets the ambiguity to acquire the property of Object class..

2. In java every class has a constructor, if we write it explicitly or not at all. The first statement is calling `super()` to invoke the supper class constructor. If the class has more than one super class, it gets confused.

So when one class extends from more than one super class, we get compile time error.

Share Edit Follow

edited Jul 27 '19 at 10:12
Tomerikoo
12.9k ● 11 ● 31 ● 43

answered Jul 27 '19 at 10:02
raghu.g.b Raghu
31 ● 1

3

Since this topic is not close I'll post this answer, I hope this helps someone to understand why java does not allow multiple inheritance.

Consider the following class:

```
public class Abc{

    public void doSomething(){

    }

}
```

In this case the class Abc does not extends nothing right? Not so fast, this class implicit extends the class Object, base class that allow everything work in java. Everything is an object.

If you try to use the class above you'll see that your IDE allow you to use methods like: `equals(Object o)`, `toString()`, etc, but you didn't declare those methods, they came from the base class `Object`

You could try:

```
public class Abc extends String{

    public void doSomething(){

    }

}
```

This is fine, because your class will not implicit extends `Object` but will extends `String` because you said it. Consider the following change:

```
public class Abc{
    public void doSomething(){

    }

    @Override
    public String toString(){
        return "hello";
    }

}
```

Now your class will always return "hello" if you call toString().

Now imagine the following class:

```
public class Flyer{

    public void makeFly(){

    }

}

public class Bird extends Abc, Flyer{

    public void doAnotherThing(){

    }

}
```

Again class `Flyer` implicit extends Object which has the method `toString()`, any class will have this method since they all extends `Object` indirectly, so, if you call `toString()` from `Bird`, which `toString()` java would have to use? From `Abc` or `Flyer`? This will happen with any class that try to extends two or more classes, to avoid this kind of "method collision" they built the idea of **interface**, basically you could think them as an **abstract class that does not extends Object indirectly**. Since they are **abstract** they will have to be implemented by a class, which is an **object** (you cannot instanciate an interface alone, they must be implemented by a class), so everything will continue to work fine.

To differ classes from interfaces, the keyword **implements** was reserved just for interfaces.

You could implement any interface you like in the same class since they does not extends anything by default (but you could create a interface that extends another interface, but again, the "father" interface would not extends Object"), so an interface is just an interface and they will not suffer from "**methods signature colissions**", if they do the compiler will throw a warning to you and you will just have to change the method signature to fix it (signature = method name + params + return type).

```
public interface Flyer{

    public void makeFly(); // <- method without implementation

}

public class Bird extends Abc implements Flyer{

    public void doAnotherThing(){

    }

    @Override
    public void makeFly(){ // <- implementation of Flyer interface

    }

    // Flyer does not have toString() method or any method from class Object,
    // no method signature collision will happen here

}
```

Share Edit Follow

answered Jun 11 '16 at 5:20

Murillo Ferreira

1,303 ● 1 ● 15 ● 28

1

in simple manner we all know, we can inherit(extends) one class but we can implements so many interfaces.. that is because in interfaces we don't give an implementation just say the functionality. suppose if java can extends so many classes and those have same methods.. in this point if we try to invoke super class method in the sub class what method suppose to run??, compiler get confused example:- try to multiple extends but in interfaces those methods don't have bodies we should implement those in sub class.. try to multiple implements so no worries..

Share Edit Follow

answered Sep 19 '19 at 10:16

hasanga lakdinu

39 ● 3

1

For example two class A,B having same method m1(). And class C extends both A, B.

```
class C extends A, B // for explaining purpose.
```

Now, class C will search the definition of m1. First, it will search in class if it didn't find then it will check to parents class. Both A, B having the definition So here ambiguity occur which definition should choose. So JAVA DOESN'T SUPPORT MULTIPLE INHERITANCE.

Share Edit Follow

answered May 13 '18 at 6:57
Pooja Khatri
355 ● 1 ● 11

how about making the java compiler gives compiler if the same methods or variables defined in both parent class so that we can change the code... – siluveru kiran kumar Aug 9 '19 at 6:06

1

Because an interface is just a contract. And a class is actually a container for data.

Share Edit Follow

answered Mar 25 '10 at 12:42
Snake
7,600 ● 5 ● 43 ● 67

The fundamental difficulty with multiple inheritance is the possibility that a class might inherit a member via multiple paths which implement it differently, without providing its own overriding implementation. Interface inheritance avoids this because the only place interface members can be implemented is in classes, whose descendants will be limited to single inheritance. – supercat Jun 16 '13 at 19:58

0

[the image explaining the problem with multiple inheritances.](#) **Just see the above Image**

> what is the inherited member of the derived class? it is still private or publically available in the derived class? for not getting this type of problem in java they remove the multiple inheritances. this image is a simple example of an object-oriented programming problem.

Share Edit Follow

edited Aug 31 at 16:55

answered Aug 31 at 16:49
Meet
1 ● 1

0

Consider a scenario where Test1, Test2 and Test3 are three classes. The Test3 class inherits Test2 and Test1 classes. If Test1 and Test2 classes have same method and you call it from child class object, there will be ambiguity to call method of Test1 or

Test2 class but there is no such ambiguity for interface as in interface no implementation is there.

Share Edit Follow

answered Apr 21 '15 at 9:41
Nikhil Kumar
2,158 ● 3 ● 17 ● 24

0

Take for example the case where Class A has a getSomething method and class B has a getSomething method and class C extends A and B. What would happen if someone called C.getSomething? There is no way to determine which method to call.

Interfaces basically just specify what methods a implementing class needs to contain. A class that implements multiple interfaces just means that class has to implement the methods from all those interfaces. Whci would not lead to any issues as described above.

Share Edit Follow

edited May 20 '13 at 19:10

answered Mar 25 '10 at 13:08
John Kane
4,330 ● 1 ● 24 ● 39

2 "*What would happen if someone called C.getSomething.*" It is an error in C++. Problem solved. – curiousguy May 19 '13 at 16:14

That was the point... that was a counter example that I thought was clear. I was pointing out that there is no way to determine which get something method should be called. Also as a side note, the question was related to java not c++ – John Kane May 20 '13 at 19:09

I am sorry I do not get what your point is. Obviously, there are ambiguity in some cases with MI. How is that a counter example? Who claimed that MI never result in ambiguity? "*Also as a side note, the question was related to java not c++*" So? – curiousguy May 20 '13 at 23:20

I was just trying to show why that ambiguity exists and why it doesnt with interfaces. – John Kane May 20 '13 at 23:59

Yes, MI can result in ambiguous calls. So can overloading. So Java should remove overloading? – curiousguy May 21 '13 at 0:19

0

Java does not support multiple inheritance , multipath and hybrid inheritance because of ambiguity problem:

```
 Scenario for multiple inheritance: Let us take class A , class B , class C.
class A has alphabet(); method , class B has also alphabet(); method. Now class
C extends A, B and we are creating object to the subclass i.e., class C , so  C
ob = new C(); Then if you want call those methods ob.alphabet(); which class
method takes ? is class A method or class B method ?  So in the JVM level
ambiguity problem occurred. Thus Java does not support multiple inheritance.
```

multiple inheritance

> ***Reference Link:***
> https://plus.google.com/u/0/communities/102217496457095083679

Share Edit Follow edited Jan 23 '17 at 14:21 answered Jan 23 '17 at 14:03 user7445071

-1

*** This is a simple answer since I'm a beginner in Java ***

Consider there are three classes `X`, `Y` and `Z`.

So we are inheriting like `X extends Y, Z` And both `Y` and `Z` is having a method `alphabet()` with same return type and arguments. This method `alphabet()` in `Y` says to *display first alphabet* and method alphabet in `Z` says *display last alphabet*. So here comes ambiguity when `alphabet()` is called by `X`. Whether it says to display *first* or *last* alphabet??? So java is not supporting multiple inheritance. In case of Interfaces, consider `Y` and `Z` as interfaces. So both will contain the declaration of method `alphabet()` but not the definition. It won't tell whether to display first alphabet or last alphabet or anything but just will declare a method `alphabet()`. So there is no reason to raise the ambiguity. We can define the method with anything we want inside class `X`.

So in a word, in Interfaces definition is done after implementation so no confusion.

Share Edit Follow edited Jan 5 '18 at 8:27 Cà phê đen 1,811 ● 2 ● 17 ● 18 answered Jan 5 '18 at 6:48 AJavaLover 1

-1

It is a decision to keep the complexity low. With hybrid inheritance, things would have been more complicated to implement, and anyways what is achievable by multiple inheritances is also with other ways.

Share Edit Follow answered Jun 19 at 20:13 hema 3 ● 2

There are more concrete answers. I'm thinking specifically of C++'s "diamond problem" – ahoffer Jun 20 at 1:50