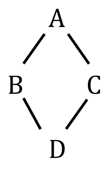
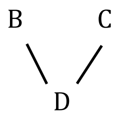
The diamond problem: multiple inheritance



Google “diamond problem” and you will get a bunch of websites that talk about the diamond problem in OO languages, showing a diamond like that drawn to the right. It shows classes or interfaces (Java terminology) `A`, `B`, `C`, and `D`, with (1) `B` and `C` extending or implementing `A` and (2) `D` extending or implementing both `B` and `C`. The diamond problem has to do with multiple inheritance. If both `B` and `C` declare a method `m` and `D` calls `m`, which method should be called, the one in `B` or the one in `C`? This question has to be answered in some unambiguous way.

Multiple inheritance is an issue not just in Java but in many OO languages like C++, Common Lisp, C#, Eiffel, Go, OCaml, Perl, Python, Ruby, and Scala. Each OO language solves the ambiguity in some way.



We show you how Java handles the diamond problem in Java, talking a bit about its history. For many Java cases, it’s not a *diamond* problem, it’s a *Vee* problem. Only `B`, `C`, and `D` are needed to explain some issues. At the end, we show you a diamond problem.

1. B, C, and D are classes

The program to the right is not a legal Java program because class `D` may extend only one class, and here it extends both `B` and `C`. But if it were legal, it would be ambiguous because in class `D`, it is not known which method `m` to call, the one inherited from class `B` or the one inherited from class `C`. Java avoids the multiple inheritance problem for classes by allowing a class to extend only one other class.

```
class B { int m() {return 0;} }
class C { int m() {return 1;} }
class D extends B, C {
   void p() {System.out.println(m());}
}. // not legal Java
```

2. B, C are interfaces and D is a class, in version 7 or less

Class `D` can implement many interfaces. To the right, it implements both `B` and `C`. Further, in Java 7 and earlier, all methods in an interface are abstract, and any non-abstract class that implements the interface must override interface’s abstract methods. The interface defines only the *syntax* of calls on a method, so there is no ambiguity.

```
interface B { int m(); }
interface C { int m(); }
class D implements B, C {
   void p() {System.out.println(m());}
   public int m() {return 5;}
}
```

3. B and D are classes and C is an interface, in version 7 or less

To the right, class `B` declares method `m` and interface `C` declares abstract `m`. Since `D` inherits `m` from `B`, `D` need not declare `m` again; it is already available. There is no ambiguity here because interface `C` defines only the syntax of calls on `m`.

Suppose `B` does not declare `m` to be public. Then there is a syntax error: inherited method `m` cannot hide public abstract method `m` in `C`. There is a choice: either have `m` in `B` be public or declare public method `m` in `D`.

```
class B { public int m() {return 0;} }
interface C { int m(); }
class D extends B implements C {
   void p() {System.out.println(m());}
}
```

4. Default interface methods in Java version 8

Version 8 of Java, released in Spring 2014, allowed default methods in interfaces, as shown to the right. Since interface `C` has a default method `m`, class `D` does not have to override `m`; if it doesn’t, the default method is used.

```
interface C { default int m() {return 1;}}
class D implements C {
   void p() {System.out.println(m());}
}
```

Ah, but now we have a problem. With the classes and interfaces shown to the right, what method `m` will be called in method `D.p`? The one in class `B` or the one in interface `C`?

For backward compatibility (a Java 7 program should run in Java 8) the Java designers ruled that the method in `B` should be called: the superclass has precedence over the interface.

```
class B { public int m() {return 0;} }
interface C { default int m() {return 1;}}
class D extends B implements C {
   void p() {System.out.println(m());}
}
```

The diamond problem: multiple inheritance

5. How does one call the interface default method?

In method D.p, how does one call inherited default method C.m? Use C.**super**.... to designate the implemented interface C whose method is to be called:

    C.super.m();

Here, there is no ambiguity. In method p, a call m() calls method m in inherited from class B, and a call C.**super**.m() calls default method m inherited from interface C.

```
class B { public int m() {return 0;} }

interface C { default int m() {return 1;}}

class D extends B implements C {
   void p() {
      System.out.println(C.super.m());
   }
}
```

6. No class extension, no multiple interfaces with a default

If class D does not extend a class and implements two interfaces that both have a default method m, as shown to the right, the program has a syntax error and won't compile. Even if method m in one of the interfaces is abstract, it's still a syntax error.

```
interface C1 { default int m() {return 1;}}
interface C2 { default int m() {return 2;}}
class D implements C1, C2 {
   …
} // syntax error: won't compile
```

7. With a class extension, multiple interfaces with defaults are OK

In the program to the right, D extends class B and implements interfaces C1 and C2. All three of them —B, C1, and C2— declare method m. The return statement in method D.p shows how to call all three of the inherited methods.

```
class B { public int m() {return 0;} }
interface C1 {default int m() {return 1;} }
interface C2 {default int m() {return 2;} }
class D extends B implements C1, C2 {
   public int p() {
      return m() + C1.super.m() + C2.super.m();
   }
}
```

8. Example of backward compatibility

One reason to prefer a method in the superclass over the default method in the interface is to maintain backward compatibility. Here's one example of why this choice was made. In Java 7, interface java.util.List had no abstract method to sort a list. In Java 8, to make life easier for programmers, this default method was to interface List:

    default void sort(Comparator<? super E> c)

It can be used to sort any List. For the Java 7 program shown on the right to work as it did in Java 7, B.sort had to be called from method p, even though List has a default method sort.

```
class B {
   …
   public void sort(…) {…}
   …
}
class D extends B implements List {
   void p() {… sort(…); … }
}
```

9. A diamond problem

The program to the right has the A-B-C-D diamond. We discuss variations of it. It might help to try these out in DrJava, calling methods from the Interactions Pane.

```
interface A {default int m() {return 1;} }
interface B extends A { }
interface C extends A { }
class D implements B, C {
   void p() { System.out.println(m());}
}
```

1. The program compiles, and execution of method p in D prints 1. The call on m can be replaced by B.**super**.m () and C.**super**.m ().

2. Put this method in B: **default int** m() { **return** 2; }.
The program remains legal. A call of method p in D prints 2 —the declaration in B overrides that in A. You can use B.**super**.m ()in method p, but C.**super**.m () won't work —it results in the error message: *bad type qualifier C in default super call method m() is overridden in B.*

3. Put this method in both B and C: **default int** m () { **return** 2; }. The program is syntactically incorrect. The error message is: *class D inherits unrelated defaults for m() from types B and C.*

4. Put this abstract method in B: **int** m(); . You get a syntax error; D does not override this abstract method.