Overload and hide methods in Java

Asked 11 years, 6 months ago Active 2 years, 4 months ago Viewed 17k times

14

5

i have an abstract class BaseClass with a public `insert()` method:

```
public abstract class BaseClass {

 public void insert(Object object) {
  // Do something
 }

}
```

which is extended by many other classes. For some of those classes, however, the `insert()` method must have additional parameters, so that they instead of overriding it I overload the method of the base class with the parameters required, for example:

```
public class SampleClass extends BaseClass {

 public void insert(Object object, Long param){
  // Do Something
 }

}
```

Now, if i instantiate the SampleClass class, i have two `insert()` methods:

```
SampleClass sampleClass = new SampleClass();
sampleClass.insert(Object object);
sampleClass.insert(Object object, Long param);
```

what i'd like to do is to hide the `insert()` method defined in the base class, so that just the overload would be visible:

```
SampleClass sampleClass = new SampleClass();
sampleClass.insert(Object object, Long param);
```

Could this be done in OOP?

java oop overloading

Share Edit Follow

edited May 12 '19 at 14:07

asked Apr 6 '10 at 23:58

Mark

Dan Is Fiddling By Firelight
5,770 ● 16 ● 73 ● 120

59.4k ● 43 ● 114 ● 154

4 Answers

Active | Oldest | Votes

21

There is no way of *hiding* the method. You can do this:

```
@Override
public void insert(Object ob) {
  throw new UnsupportedOperationException("not supported");
}
```

but that's it.

The base class creates a contract. All subclasses are bound by that contract. Think about it this way:

```
BaseObject b = new SomeObjectWithoutInsert();
b.insert(...);
```

How is that code meant to know that it doesn't have an `insert(Object)` method? It can't.

Your problem sounds like a design problem. Either the classes in question shouldn't be inheriting from the base class in question or that base class shouldn't have that method. Perhaps you can take `insert()` out of that class, move it to a subclass and have classes that need `insert(Object)` extend it and those that need `insert(Object, Object)` extend a different subclass of the base object.

Share Edit Follow

answered Apr 7 '10 at 0:04

cletus
586k ● 160 ● 893 ● 936

one more edition to your solution: add @Deprecated annotation to the overrided method which should be hided. Why? because then when developer will try to call this method, he will see deprecated message, but not catch exception in runtime. – Andriy Antonov Jul 6 '18 at 10:23

6

I don't believe there's a clean way to completely hide an inherited method in Java.

In cases like this, if you absolutely can't support that method, I would probably mark that method as @Obsolete in the child class, and have it throw a NotImplementedException (or whatever the equivalent exception is in Java), to discourage people from using it.

In the end, if you inherit a method that does not make sense for your child class, it could be that you really shouldn't inherit from that base class at all. It could also be that the base class is poorly designed or encompasses too much behavior, but it might be worth considering your class hierarchy. Another route to look at might be composition, where your class has a private instance of what used to be the base class, and you can choose which methods to expose by wrapping them in your own methods. (Edit: if the base class is abstract, composition might not be an option...)

Share Edit Follow edited Apr 7 '10 at 0:10

answered Apr 7 '10 at 0:03

Andy White
82.5k ● 47 ● 171 ● 207

2

As Cletus points out, this is really a design problem, in that you are trying to create a child class that does not obey the contract of its parent class.

There are rare circumstances where working around this by e.g. throwing an exception might be desirable (or at least an acceptable compromise -- for example, the Java Collections Framework) but in general it's a sign of poor design.

You may wish to read up on the Liskov substitution principle: the idea that (as Wikipedia puts it) "if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program". By overriding a method to throw an exception, or hiding it any other way, you're violating this principle.

If the contract of the base class' method was "inserts the current object, or throws an exception" (see e.g. the JavaDoc for Collection.add()) then you could argue you're not violating LSP, but if that is unexpected by most callers you may want to rethink your design on these grounds.

Share Edit Follow

answered Apr 7 '10 at 0:46

Cowan
35.9k ● 11 ● 65 ● 63

1

This sounds like a badly designed hierarchy -

If no default exists and the user shouldn't call the method at all you can mark the method as `@Deprecated` and throw an `UnsupportedOperationException` as other

posters have noted. **However** - this is really only a runtime check. `@Deprecated` only throws a compiler warning and most IDEs mark it in some way, but there's no compile time prevention of this. It also really sucks because it's possible to get the child class as a parent class reference and call the method on it with no warning that it's "bad" at all. In the example below, there won't be any indication until runtime that anything's wrong.

Example:

```
// Abstract base builder class
public abstract class BaseClassBuilder {
    public final doBuild() {
        BaseClass base = getBase();
        for (Object obj : getObjects() {
            base.insert(obj);
        }
    }
    protected abstract BaseClass getBase();
    protected abstract Object[] getObjects();
}

// implementation using SampleClass
public class SampleClassBuilder extends BaseClassBuilder {

    @Override
    protected BaseClass getBase() {
        return new SampleClass();
    }

    @Override
    protected Object[] getObjects() {
        Object[] obj = new Object[12];
        // ...
        return obj;
    }
}
```

However, if a sensible default exists, you could mark the inherited method as final and provide the default value inside of it. This handles both the bad hierarchy, and it prevents the "unforseen circumstances" of the above example.

Example:

```
public abstract class BaseClass {
    public void insert(Object object) {
        // ...
    }
}

public class SampleClass extends BaseClass {

    public static final Long DEFAULT_PARAM = 0L;

    public final void insert(Object object) {
        this.insert(object, DEFAULT_PARAM);
    }
```

```
    public void insert(Object object, Long param) {
        // ...
    }
}
```

Share Edit Follow

answered Apr 7 '10 at 0:55

Nate

16.3k ● 5 ● 45 ● 58