

Re: Static methods overridden !!

Source: <http://coding.derkeiler.com/Archive/Java/comp.lang.java.programmer/2007-03/msg02977.html>

- *From:* "Chris Uppal" <chris.uppal@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Mon, 26 Mar 2007 09:35:09 +0100
-

Ravi wrote:

I am not talking about polymorphism here, just overriding.

In Java overriding /is/ polymorphism. That's to say, the only use of the word "overriding" that's allowed (if you want to talk be understood by other Java programmers) is to mean the instance-side re-implementation of inherited methods by subclasses (or in implementations of interfaces).

As you know, Java's static methods don't work like that, so the word "overriding" is simply not applicable here.

Probably the best way to think about what's happening is to take it in steps.

[WARNING: the following is a simplified view, and is inaccurate in small ways that I'll explain at the end of this post]

First off, as you also know, you can invoke any (public) static method from any place by using the syntax:
`fully.qualified.ClassName.staticMethod();`

When it comes right down to it -- i.e. in terms of how the compiler treats static methods -- that's the /only/ way to invoke a static method. But the compiler does allow a number of convenient abbreviations.

One, which has already been discussed, is the ill-conceived idea that you can replace the name of the class by some expression with the (statically determined) type which is the name of the class.

Another abbreviation is that if you have an
`import fully.qualified.ClassName;`
or
`fully.qualified.*;`
statement in your Java source, then you can refer to the class by its short name, and thus invoke the static method by the shorter expression:
`ClassName.staticMethod();`

Re: Static methods overridden !!

Yet another abbreviation is that if the method name is "in scope" then you don't need the class name at all, you can just write:
staticMethod();

Now, the concept of being in scope is interesting. It applies everywhere within the body of the class where the static method is defined. And it /also/ applies throughout the bodies of any subclasses of that class. Hence, within a subclass, you can also write

```
staticMethod();
```

and the name will be resolved to:

```
fully.qualified.ClassName.staticMethod();
```

at compile time (remember the above warning — that assertion is technically incorrect, although it does capture the /intended/ idea of what's happening).

A further refinement of the idea of scoping is that if you have a class, Super, which defines a static method, aMethod(), and you have a subclass Sub, which does /not/ define a method of the same name and signature, then you can invoke Super.aMethod() with the method call expression:

```
Sub.aMethod();
```

And "aMethod" is resolved in the scope defined by Sub, which includes Super.aMethod().

You /can/ call that "inheritance" if you like — it's certainly not an abuse of the word — but it's important to realise that what is being inherited is the just the scope in which names are resolved (at compile time). The kind of inheritance we see here is completely different from the kind of polymorphic, OO, inheritance which is used for (non-private, non-final) instance methods. But there is no overriding involved at all (by definition of the word "override" as it is used in Java).

All OK so far ? I hope so, because it's a reasonably simple picture, and it does reflect how Java is intended to work and how we (as programmers) are intended to think about it. If you aren't interested in details, then stop reading now, because the rest of this post will just add confusion without making you better able to understand ordinary Java code.

Right, I said that the above was technically inaccurate; here's how. I said that the compiler statically fills in the name of the class when it expands the abbreviation. That's true, but there are rules about /which/ class name it fills in. You might expect it to look to see what class defines the method (which it has to do anyway) and use the name of that class to expand the abbreviation. In fact, it doesn't do that (although there were bugs in the javac compiler in this area as late as JDK 1.3). When a programmer writes:
staticMethod();
the compiler uses the class where the method call is written to begin resolving the name "staticMethod()", it will find the definition of that method in some superclass, but it is not allowed to expand the abbreviation by naming that superclass explicitly — it is required (by the JLS) to expand it out to the name of the class where the lookup /began/. In this case the name of the class

Re: Static methods overridden !!

where the code which calls `staticMethod()` is defined. For instance, if you have:

```
class fully.qualified.Super
{
  static void aMethod() {}
}
class fully.qualified.Sub extends fully.qualified.Super
{
  void someCode() { aMethod(); }
}
class my.Class
extends fully.qualified.Sub
{
}
```

and you create an instance of class `my.Class`, and call the (instance-side) `someCode()` method that inherits from `fully.qualified.Sub`, then the bytecode which are executed will contain the equivalent of the Java code (after expansion by the compiler):

```
fully.qualified.Sub.aMethod();
```

even though there is no such method as `fully.qualified.Sub.aMethod()` ! That is surprising because it means that `aMethod()` is /actually/ resolved, at runtime, by a search up the inheritance hierarchy[*], despite the fact that Java's static classes are, according to the language design, always resolved /statically/. There is room for debate about why there is this odd anomaly in the specification. One hypothesis is that the language designers were just confused about static methods, and didn't really have a clear idea of what they were trying to achieve. Another hypothesis is that it's all about supporting binary compatibility[**] as classes evolve independently of each other. My own opinion is that there's a bit of truth in both hypotheses, but that the second one is a bit more true than the first ;-)

-- chris

[*] Obviously the JVM will optimise out that search -- there's no time penalty at runtime.

[**] There is a large section of the JLS which is all about allowing class files to work correctly even if they have been compiled against different versions of the other classes they mention. Or, if not always /correctly/, then at least with clearly defined behaviour, requirements, and guarantees. See the JLS for details.

.