What is this referencing?

Suppose I have this class:

```
public class class1 extends Applet implements Runnable
{
    private String s;
    private URL u;
    ...
}
```

And a second class:

```
class TS extends Thread
{
    private final class1 _$97913;
    public TS(class1 paramclass1)
    {
        this._$97913 = paramclass1;
    }
    ...
    public void PostData()
    {
        ...
        class1.access$16(this._$97913, new Socket(class1.access$17(this._$97913),
80);
        ...
    }
    ...
}
```

Can someone explain how `class1.access$16(this._$97913, new Socket(class1.access$17(this._$97913), 80);` is referencing `private URL u;` from class1?

Where does the `access$16` come from? What is this called and where can I learn more about it?

Ok, this being the result of decompiled code, is there a way to associate the numbers (`access$16` , `access$17` , etc.) to the original variable or class? From what I can see, the only way would be to do so manually (i.e. see what is being referenced where and guess that since 'this' class received a URL, then 'this' must be associated with 'that' variable)?

java decompiler

edited Apr 3 '11 at 6:41
Gabe
60.2k 5 91 169

asked Apr 2 '11 at 16:24
Josh C.
68 1 6

2 Answers

> Ok, this being the result of decompiled code, is there a way to associate the numbers (`access$16` , `access$17` , etc.) to the original variable or class? From what I can see, the only way would be to do so manually (i.e. see what is being referenced where and guess that since 'this' class received a URL, then 'this' must be associated with 'that' variable)?

The `access$x` methods are created if you access private methods or variables from a nested class (or the other way around, or from one nested class to another). They are created by the compiler, since the VM does not allow direct access to private variables.

If the decompiler lets these method calls stay in the recreated source code for the using class, it should also let the synthetic methods definitions stay in the recreated source code for the used class. If so, have a look at the class which is the receiver of the method in question (`class1` in your case), there should be such a method (`access$17`). In the code of this method you can see which real method (or variable) is accessed here.

If the decompiler removed the synthetic methods, this is either a bug, or it may be configurable. It could also be that you have to pass it all the classes at once, and then it can put in the right methods/fields everywhere - look at its documentation.

If you have *the classes before the dot* of a method call (and their superclasses, if any), you should have the methods.

From the snippet you posted, there should be a `access$16` and `access$17` method in `class1` (or is `class1` a local variable here?).

If it isn't, maybe your decompiler tried to be smarter then he should. You could have a look at the output of `javap class1` to see if the methods are there, and `javap -c class1` for the whole bytecode. Or use another decompiler.

edited Apr 2 '11 at 19:44

answered Apr 2 '11 at 18:19

Paŭlo Ebermann
47.3k 9 85 147

class1 create a new instance of TS `this.var = new xplug.SI(this);` (var is `private class1.TS var`) Unfortunately, there is no documentation available to look at. There is no `access$17` method in `class1` , and I doubt it's a bug. I think it could also be, although less likely, that I'm missing the file that contains these methods. – Josh C. Apr 2 '11 at 19:13

(a bug in the decompiler, I meant.) – Paŭlo Ebermann Apr 2 '11 at 19:19

Either way, thanks so much for your help. I'll see what I can do about it from here. – Josh C. Apr 2 '11 at 19:19

See my last edit for some more details about this. (I was interrupted when starting to type, thus only now.) – Paŭlo Ebermann Apr 2 '11 at 19:45

I saw your last edit just now and it has proved very useful. From what I can tell it seems that I may either be missing a class file or it was configured to compile this way. Here is what makes me think that: At first I decompiled the class1.class file and it had many `access$x` methods being called that were nowhere to be found. Afterwards, I got a hold of the `class1$TS.class` file. After I decompiled it again, some of the `access$x` references where replaced with the correct method/variable. So it seems that I may be missing an additional class file. Once again, thanks for your help. – Josh C. Apr 2 '11 at 22:04

Is this the result of decompiling java?

It looks like a synthetic method created to allow outer and inner classes access to one another's private fields or methods.

> The Java compiler must create synthetic methods on nested classes when their attributes specified with the private modifier are accessed by the enclosing class. The next code sample indicates this situation.
>
> ...
>
> As the above screen snapshot indicates, a synthetic method with the name access$100 has been created on the nested class NestedClass to provide its private String to the enclosing class. Note that the synthetic method is only added for the single private attribute of the NestedClass that the enclosing class accesses.

answered Apr 2 '11 at 16:27

Mike Samuel
73.6k 16 134 180

`It looks like a synthetic method` It looks horrendous. FTFY – Finbarr Apr 2 '11 at 16:32

Yes, this is from decompiled code. And thanks for the quick answer. That answers most of my questions I had about this. – Josh C. Apr 2 '11 at 16:34

In all seriousness, definitely looks like a synthetic method. +1 – Finbarr Apr 2 '11 at 16:35

@Josh: If it does not answer all questions, add the remaining questions to your question (there is an edit button). When all your questions are answered, mark the best answer as "accepted". – Paŭlo Ebermann Apr 2 '11 at 16:45

@Paŭlo: Thanks, I have done as you suggested. – Josh C. Apr 2 '11 at 17:06