How Many Trivial Getter Methods Does Java Have?

Elliott Hughes
enh@acm.org

July 3, 2002

Abstract

A recent SIGPLAN Notice [5] repeated a claim [4] that "accessor methods used to obtain the value of a field (getter methods) account for about one fourth of the total number of methods in the standard `java.*` package source classes". This article examines that claim, and examines the variety of methods that are nowadays considered getter methods.

1 Getter methods

1.1 A misunderstanding

Although the motivation for making fields private is supposedly well known, experience suggests that it's all too easy for the learner to fall into writing Cargo Cult Object-Oriented programs. Like the pacific islanders who, when they saw airplanes landing on airstrips, assumed that if you build what looks like an airstrip, planes will come, many learners seem to misunderstand the idea of making fields private.

One reason for this is that when the practice of declaring fields private is exalted, it's necessary to explain get and set methods because they're bound to come across them. The learner promptly invents a coping strategy centered around get and set methods that lets them program just as they used to.

Even if one explains to the learner that they should only write get and set methods if they *really* need access to fields, it's not easy to convey how one recognizes such cases. The students, who don't yet have much OO experience, assume that access is needed to most if not all of the fields in their class, because they're thinking – if not explicitly in terms of C structs and functions – of objects as collections of state and behavior rather than as service providers.

They've listened to what they've been told, missed the point entirely, and come away thinking that all that's needed to ensure they'll write good programs – and most of them do want to write good programs – is to type

```
private int length;

public int getLength() {
    return this.length;
}
```

in place of

```
public int length;
```

In many cases, this is a step forwards in moving them away from writing structs and global functions, but the poor student is more likely to find themselves writing pure state 'objects' than the pure behavior ones no-one

thought to point out are the real motivation for private fields.

This may not be as unfortunate as attacking the Japanese army and pitting your wood-carved 'guns' against their genuine firearms, but it's unfortunate all the same.

1.2 The case against getters

Some people resent writing all these getters. They consider the extra keystrokes aren't worth the benefit of slightly constraining the kind of operations that can be performed on an object. They say that if unconstrained access turns out to be a problem, you can just refactor the code. This is true, though in this case it's a lot easier to have not made the mistake in the first place.

The other common claim is that field accessor syntax is more natural, though it's not clear in what sense, other than "more similar to C structs". Because `this` is implicit in most languages, internal references don't tend to contain many dots anyway.

There's also a claim of inefficiency, but it's highly likely that any getter worth inlining will be inlined. Even the so-called non-inlining compiler from the Self project (described in chapter 4 of Hölzle's thesis [2]) performed this optimization, which was a good job because Self implemented all instance variable accesses as dynamically-dispatched method calls. More to the point, though, why would a call to another object's getter method be a performance hot-spot?

2 Non-trivial getters

In reality, though, few getter methods are like the trivial kind shown above. Three reasons for this are:

1. Not everything an object might want to return from a getter method is conveniently located in one of its fields. Sometimes the answer has to be worked out, and that means the getter method has to be more complicated.

2. An object doesn't always want to return the value it's using. The getLength example above is fine, because the invoker only has a copy of the int to play with, but imagine a GUI component that offers a getSize method. If the component stores its size in an object that offers no way of modifying it, it's safe to return that object. It's more likely, though, that the object *can* be modified. The Dimension class often used for this purpose by Java programmers, for example, can be modified. A component shouldn't give other objects access to a modifiable object: what would happen if the recipient decided to modify that object? C++ programmers perhaps don't see this distinction quite as clearly because C++ offers pass-by-value. A Java programmer must explicitly copy an object and return the copy.

3. An object should expose behavior, not state. This is sometimes called the "tell, don't ask" principle [3]. A common mistake amongst inexperienced OO programmers is to think that the methods their classes need are those that expose enough state for them to write the computations they need in all the places they need them, rather than making these computations something that the class knows how to do. As Kent Beck wrote [1]: "Making an accessing method public should be done only when you can prove to yourself that there is no way for the object to do the job itself".

These are all reasons why one doesn't want to see many trivial getters, but a shorthand notation for writing them addresses none of

these issues. What's needed is to write more suitable methods instead.

3 Setter methods

Setter methods cause far less trouble and resentment than getter methods, perhaps because setter methods almost always have to do more work than a simple assignment, even if they only need to check that the new value is acceptable.

This lack of controversy is odd in light of setter methods' greater potential for trouble-making. As Kent Beck said directly following the last sentence I quoted, "Making a setting method public requires even more soul-searching, since it gives up even more of an object's sovereignty".

4 How many trivial getters does Java really have?

4.1 Earlier work

The Roses [4] counted matches for the following regular expressions to find the number of public methods and the number of public getter methods respectively:

```
" +public .*("
" +public .* get.*("
```

There are a few things that aren't quite right here. To begin with, it's not clear why the Roses only cared about public getter methods. One could imagine an argument that trivial private getter methods were more questionable than trivial public ones. It's also not clear why they weren't interested in those Boolean getter methods whose names begin `is` rather than `get`. They also fail to give special treatment to `synchronized` methods that arguably do more than just return the value of a field. These points can all be assumed to be explicit decisions on their part rather than mistakes.

More seriously, their regular expression for recognizing public getter methods is wrong. It matches `static` class methods whose name begins with 'get' such as `getInstance` and others used to implement the Singleton pattern. In fact, it matches all methods whose names begin with 'get' even if they aren't getters in the trivial sense they had in mind. The shortest examples of these are the `get` methods in classes implementing data structures such as `Hashtable` and `Vector`, but it's more of a problem than that: because they don't examine the code within a suspected getter, they can't say whether or not the getter could be removed given their system of read-only fields.

4.2 Aside: what makes a getter?

This kind of confusion about what 'get' methods do suggests that maybe there is something in the claim that such method names are unnatural, but on the other hand it may simply be that it's the only naming pattern that every programmer knows.

If only we had the shared vocabulary to distinguish between actual getters (also known, along with setters, as accessor methods) and such things as:

- 'fetchers' such as `AppletContext.getImage` which cause an image to be downloaded and decoded rather than simply return the appropriate image from their infinite bag of all possible images.

- 'fillers' that made up for older garbage collectors that had problems with lots of short-lived objects. These methods take an 'out' pa-

rameter which they modify, such as `Component.getBounds(Rectangle)`.

- 'translators' that convert the object into an alternative form such as `String.getBytes` — these methods are named using the prefix 'as' rather than 'get' in Smalltalk.

- 'pure function' methods that perform a calculation with no side-effects and return the result such as `TextArea.getPreferredSize(int, int)`.

- methods that exist because a suitable class doesn't. You could imagine a `Font.getWidth(String)` method to return the width of a string in a particular font, or you could imagine it with the receiver and argument the other way round (i.e. a string method that takes a font). But without a new class uniting a font and a string (call it `StyledText`) you can't have a 'get' method that takes no arguments.

- 'promise' methods which actually use a one-shot variant of the Observer pattern to return their answer, rather than returning it immediately, such as `Image.getWidth(ImageObserver)`.

- methods that read external state such as `Robot.getPixelColor(int, int)`.

- 'pass-through' methods invented as short-cuts such as `Applet.getImage` which actually does nothing more than invoke the same method in the instance of `AppletContext` which would have been available through `Applet.getAppletContext`.

I'd suggest that `.get` has become every bit as natural to many programmers as `.` was to their forebears, perhaps to such an extent that almost any method can be called a getter if it doesn't seem to involve much behavior.

4.3 How do we get the right numbers?

Several of the problems with the Roses' scheme could be solved by using more complicated regular expressions or a longer shell pipeline. For example, `grep -v` would exclude the `static` methods, and a slight change to the regular expression would insist that a closing parenthesis appear straight after the opening parenthesis. But none of this helps us recognize whether or not a getter method is of the trivial kind that could be replaced by a read-only field, one that corresponds to a simple field access.

Luckily, the JDK includes a utility that will let us recognize trivial getters. The `javap` command disassembles Java class files. A simple Perl script (figure 1) can read the output, counting methods and examining the disassembly of the possible getter methods it finds to see how many of them are actually of the trivial "return a field" kind.

The instruction sequence for such a method consists of the `aload_0` instruction to load `this` followed by `getfield` *field-number* to load the field value, and the return instruction appropriate to the type of the field.

There aren't any special-case shorthand forms of the `getfield` instruction, as there are for loading constants or local variables, so the instruction sequence is constant-length.

4.4 What are the right numbers?

Table 1 shows the total number of methods, number of 'getter' methods (as defined by the criteria mentioned above), and number of trivial getter methods (those whose implementation is akin to public field access) for a variety of different JDKs.

Codebase	All methods	Getters	% of all methods	Trivial getters	% of getters	% of all methods
Compaq Tru64 1.1.8	13635	2064	15%	648	31%	5%
(without `sun*` classes)	6446	1047	16%	388	37%	6%
1.2.2 rt.jar `java.*`	9820	1531	16%	702	46%	7%
1.2.2 AWT/Swing	15784	2900	18%	1086	37%	7%
Apple 1.3.1 update 1 classes.jar	21148	1783	8%	772	43%	4%

Table 1: Methods, getter methods and trivial getter methods found in a variety of JDKs.

As can be seen, the proportions stay pretty much the same both in different parts of the JDK and across numerous versions. Getters, as best we can recognize them, account for about 15% of all methods, but of those only 40% are trivial. So 6% of the methods in the JDK classes are trivial getter methods, and that proportion isn't rising. In a codebase as well-documented as the JDK, it's easy to imagine that the lines of comment for these 6% of methods far outnumber the lines of code. If someone were to come up with a proposal for saving effort there, that *would* be interesting.

5 Conclusion

Trivial getter methods aren't very common, certainly not as common as claimed elsewhere [4]. This is a reason why special linguistic support for public getter methods is unnecessary. More importantly, good object-oriented programs should focus on behavior, not state. This is a reason why special linguistic support for public getter methods is a bad idea.

Acknowledgements

Thanks to Martin Dorey, Phil Norman and Ed Porter for their comments on earlier drafts.

References

[1] Kent Beck. To Accessor or Not to Accessor? Smalltalk Report, June 1993, reprinted in Kent Beck's Guide to Better Smalltalk, Cambridge University Press 1999.

[2] Urs Hölzle. Adaptive Optimization for Self: Reconciling High Performance With Exploratory Programming. Ph.D. Thesis, Technical Report STAN-CS-TR-94-1520, Department of Computer Science, Stanford University, 1994.

[3] The Pragmatic Programmers. Tell, Don't Ask. `http://www.pragmaticprogrammer.com/ppllc/papers/1998_05.html`

[4] Eva Rose and Kristoffer Høgsbro Rose. Java Access Protection through Typing. `http://www.informatik.fernuni-hagen.de/import/pi5/workshops/ecoop2000/final_versions/rose.pdf`

[5] Diomidis Spinellis. A Modest Proposal for Curing the Public Field Phobia. ACM Sigplan Notices, Vol.37 No.4, April 2002, pages 54-56.

```
#!/usr/bin/perl -w

# Generate an input file with:
#
# find . -name *.class | \
#   sed 's/^.\///' | sed 's/.class$//' | sed 's/\//./g' | \
#   xargs javap -c > all-bytecodes

my $methodCount = 0;
my $getterMethodCount = 0;
my $trivialGetterMethodCount = 0;
while (<>) {
    if (/^Method [A-Za-z0-9_.]+ get(\w+)\(\)/) {
        #print "get$1()\n";
        $methodCount++;
        $getterMethodCount++;
        my $instructionCount = 0;
        while (<>) {
            if (/^ +[0-9]+ (.*)/) {
                #print "  $1\n";
                $instructionCount++;
            } else {
                last;
            }
        }
        if ($instructionCount == 3) {
            $trivialGetterMethodCount++;
        }
    } elsif (/^Method /) {
        $methodCount++;
    }
}
print "Total methods: $methodCount\n";
print " of which getter methods: $getterMethodCount (" .
    ($getterMethodCount/$methodCount) . " of all methods)\n";
print " of which trivial ones: $trivialGetterMethodCount (" .
    ($trivialGetterMethodCount/$getterMethodCount) .
    " of all getters, " .
    ($trivialGetterMethodCount/$methodCount) .
    " of all methods)\n";
```

Figure 1: The Perl Script used to generate the statistics.