JAVA TOOLBOX

By Allen Holub, JavaWorld
SEP 5, 2003 12:00 AM PST

Why getter and setter methods are evil

Make your code more maintainable by avoiding accessors

I didn't intend to start an "is evil" series, but several readers asked me to explain why I mentioned that you should avoid get/set methods in last month's column, "Why extends Is Evil."

Though getter/setter methods are commonplace in Java, they are not particularly object oriented (OO). In fact, they can damage your code's maintainability. Moreover, the presence of numerous getter and setter methods is a red flag that the program isn't necessarily well designed from an OO perspective.

This article explains why you shouldn't use getters and setters (and when you can use them) and suggests a design methodology that will help you break out of the getter/setter mentality.

[**Also on InfoWorld: JDK 19: The new features in Java 19**]

On the nature of design

Before I launch into another design-related column (with a provocative title, no less), I want to clarify a few things.

I was flabbergasted by some reader comments that resulted from last month's column, "Why extends Is Evil" (see Talkback on the article's last page). Some people believed I argued that object orientation is bad simply because `extends` has problems, as if the two concepts are equivalent. That's certainly not what I *thought* I said, so let me clarify some meta-issues.

This column and last month's article are about design. Design, by nature, is a series of trade-offs. Every choice has a good and bad side, and you make your choice in the context of overall criteria defined by necessity. Good and bad are not absolutes, however. A good decision in one context might be bad in another.

If you don't understand both sides of an issue, you cannot make an intelligent choice; in fact, if you don't understand all the ramifications of your actions, you're not designing at all. You're stumbling in the dark. It's not an accident that every chapter in the Gang of Four's *Design Patterns* book includes a "Consequences" section that describes when and why using a pattern is inappropriate.

[REGISTER NOW for CIO 100: Symposium & Awards Conference, August 15-17]

Stating that some language feature or common programming idiom (like accessors) has problems is not the same thing as saying you should never use them under any circumstances. And just because a feature or idiom is commonly used does not mean you *should* use it either. Uninformed programmers write many programs and simply being employed by Sun Microsystems or Microsoft does not magically improve someone's programming or design abilities. The Java packages contain a lot of great code. But there are also parts of that code I'm sure the authors are embarrassed to admit they wrote.

By the same token, marketing or political incentives often push design idioms. Sometimes programmers make bad decisions, but companies want to promote what the technology can do, so they de-emphasize that the way in which you do it is less than ideal. They make the best of a bad situation. Consequently, you act irresponsibly when you adopt any programming practice simply because "that's the way you're supposed to

do things." Many failed Enterprise JavaBeans (EJB) projects prove this principle. EJB-based technology is great technology when used appropriately, but can literally bring down a company if used inappropriately.

My point is that you should not program blindly. You must understand the havoc a feature or idiom can wreak. In doing so, you're in a much better position to decide whether you should use that feature or idiom. Your choices should be both informed and pragmatic. The purpose of these articles is to help you approach your programming with open eyes.

Data abstraction

A fundamental precept of OO systems is that an object should not expose any of its implementation details. This way, you can change the implementation without changing the code that uses the object. It follows then that in OO systems you should avoid getter and setter functions since they mostly provide access to implementation details.

To see why, consider that there might be 1,000 calls to a `getX()` method in your program, and each call assumes that the return value is of a particular type. You might store `getX()`'s return value in a local variable, for example, and that variable type must match the return-value type. If you need to change the way the object is implemented in such a way that the type of X changes, you're in deep trouble.

If X was an `int`, but now must be a `long`, you'll get 1,000 compile errors. If you incorrectly fix the problem by casting the return value to `int`, the code will compile cleanly, but it won't work. (The return value might be truncated.) You must modify the code surrounding each of those 1,000 calls to compensate for the change. I certainly don't want to do that much work.

One basic principle of OO systems is *data abstraction*. You should completely hide the way in which an object implements a message handler from the rest of the program. That's one reason why all of your instance variables (a class's nonconstant fields) should be `private`.

If you make an instance variable `public`, then you can't change the field as the class evolves over time because you would break the external code that uses the field. You don't want to search 1,000 uses of a class simply because you change that class.

This implementation hiding principle leads to a good acid test of an OO system's quality: Can you make massive changes to a class definition—even throw out the whole thing and replace it with a completely different implementation—without impacting any of the code that uses that class's objects? This sort of modularization is the central premise of object orientation and makes maintenance much easier. Without implementation hiding, there's little point in using other OO features.

Getter and setter methods (also known as accessors) are dangerous for the same reason that `public` fields are dangerous: They provide external access to implementation details. What if you need to change the accessed field's type? You also have to change the accessor's return type. You use this return value in numerous places, so you must also change all of that code. I want to limit the effects of a change to a single class definition. I don't want them to ripple out into the entire program.

Since accessors violate the encapsulation principle, you can reasonably argue that a system that heavily or inappropriately uses accessors simply isn't object oriented. If you go through a design process, as opposed to just coding, you'll find hardly any accessors in your program. The process is important. I have more to say on this issue at the end of the article.

The lack of getter/setter methods doesn't mean that some data doesn't flow through the system. Nonetheless, it's best to minimize data movement as much as possible. My experience is that maintainability is inversely proportionate to the amount of data that moves between objects. Though you might not see how yet, you can actually eliminate most of this data movement.

By designing carefully and focusing on what you must do rather than how you'll do it, you eliminate the vast majority of getter/setter methods in your program. *Don't ask for the information you need to do the work; ask the object that has the information to do the work for you.* Most accessors find their way into code because the designers weren't thinking about the dynamic model: the runtime objects and the messages they send to

one another to do the work. They start (incorrectly) by designing a class hierarchy and then try to shoehorn those classes into the dynamic model. This approach never works. To build a static model, you need to discover the relationships between the classes, and these relationships exactly correspond to the message flow. An association exists between two classes only when objects of one class send messages to objects of the other. The static model's main purpose is to capture this association information as you model dynamically.

Without a clearly defined dynamic model, you're only guessing how you will use a class's objects. Consequently, accessor methods often wind up in the model because you must provide as much access as possible since you can't predict whether or not you'll need it. This sort of design-by-guessing strategy is inefficient at best. You waste time writing useless methods (or adding unnecessary capabilities to the classes).

Accessors also end up in designs by force of habit. When procedural programmers adopt Java, they tend to start by building familiar code. Procedural languages don't have classes, but they do have the C `struct` (think: class without methods). It seems natural, then, to mimic a `struct` by building class definitions with virtually no methods and nothing but `public` fields. These procedural programmers read somewhere that fields should be `private`, however, so they make the fields `private` and supply `public` accessor methods. But they have only complicated the public access. They certainly haven't made the system object oriented.

Draw thyself

One ramification of full field encapsulation is in user interface (UI) construction. If you can't use accessors, you can't have a UI builder class call a `getAttribute()` method. Instead, classes have elements like `drawYourself(...)` methods.

A `getIdentity()` method can also work, of course, provided it returns an object that implements the `Identity` interface. This interface must include a `drawYourself()` (or give-me-a-JComponent-that-represents-your-identity) method. Though `getIdentity` starts with "get," it's not an accessor because it doesn't just return a field. It returns a complex object that has reasonable behavior. Even when I have an `Identity` object, I still have no idea how an identity is represented internally.

Of course, a `drawYourself()` strategy means that I (gasp!) put UI code into the business logic. Consider what happens when the UI's requirements change. Let's say I want to represent the attribute in a completely different way. Today an "identity" is a name; tomorrow it's a name and ID number; the day after that it's a name, ID number, and picture. I limit the scope of these changes to one place in the code. If I have a give-me-a-`JComponent`-that-represents-your-identity class, then I've isolated the way identities are represented from the rest of the system.

Bear in mind that I haven't actually put any UI code into the business logic. I've written the UI layer in terms of AWT (Abstract Window Toolkit) or Swing, which are both abstraction layers. The actual UI code is in the AWT/Swing implementation. That's the whole point of an abstraction layer—to isolate your business logic from a subsystem's mechanics. I can easily port to another graphical environment without changing the code, so the only problem is a little clutter. You can easily eliminate this clutter by moving all the UI code into an inner class (or by using the Façade design pattern).

JavaBeans

You might object by saying, "But what about JavaBeans?" What about them? You can certainly build JavaBeans without getters and setters. The `BeanCustomizer`, `BeanInfo`, and `BeanDescriptor` classes all exist for exactly this purpose. The JavaBean spec designers threw the getter/setter idiom into the picture because they thought it would be an easy way to quickly make a bean—something you can do while you're learning how to do it right. Unfortunately, nobody did that.

Accessors were created solely as a way to tag certain properties so a UI-builder program or equivalent could identify them. You aren't supposed to call these methods yourself. They exist for an automated tool to use. This tool uses the introspection APIs in the `Class` class to find the methods and extrapolate the existence of certain properties from the method names. In practice, this introspection-based idiom hasn't worked out. It's made the code vastly too complicated and procedural. Programmers who don't understand data abstraction actually call the accessors, and as a consequence, the code is less maintainable. For this reason, a metadata feature will be incorporated into Java 1.5 (due in mid 2004). So instead of:

```
private int property;
public int getProperty  (           ){ return property; }
public void setProperty (int value}{ property = value; }
```

You'll be able to use something like:

```
private @property int property;
```

The UI-construction tool or equivalent will use the introspection APIs to find the properties, rather than examine method names and infer a property's existence from a name. Therefore, no runtime accessor damages your code.

When is an accessor okay?

First, as I discussed earlier, it's okay for a method to return an object in terms of an interface that the object implements because that interface isolates you from changes to the implementing class. This sort of method (that returns an interface reference) is not really a "getter" in the sense of a method that just provides access to a field. If you change the provider's internal implementation, you just change the returned object's definition to accommodate the changes. You still protect the external code that uses the object through its interface.

💡   How to choose a low-code development platform