Getters/Setters - Use or Abuse?

There are many articles out there dealing with the accessor and mutator methods (getters and setters). They came together with the Object-Oriented principles. Information hiding says that the internal state of an object must be hidden from the outer world. That is, the fields it uses shouldn't be available. Yet, sometimes we have to query the state of the object, and even may want to change that state. Directly accessing the field representing the state is considered to be a bad practice (the field shouldn't be available), therefore a method must be provided to retrieve the value of the field. This method is the accessor method (the getter). Similarly, a method might be provided to change the value of the corresponding field. This method is the mutator method (the setter). Usually, the signature of these methods follow the ***<Type> get<FieldName>()*** and ***void set<FieldName>(<Type> newValue)*** notation. So we are safe, we strictly follow the OO principles, right?

Wrong!

We just mimic the OO theory. Consider the following code snipplet.

Direct access vs accessor/mutator access

```
class MyClass1 {
    public int counter;
    private int otherCounter;
    public int getOtherCounter() { return otherCounter; }
    public void setOtherCounter(int value) { otherCounter = value; }
}

class UserClass {
    public void userMethod(MyClass1 obj) {
                int val = obj.counter;
                obj.counter = 2 * val;

                int otherVal = obj.getOtherCounter();
                obj.setOtherCounter(2 * otherVal);
    }
}
```

What's the real difference between directly accessing the "counter" field, and using accessor/mutator methods for the otherCounter? Not much, isn't it? The only advantage is that by using the methods, the implementor is free to change the type of the variable later (e.g. to long). The methods will work none the less (yes, I know that you are smart enough to spot that converting the long otherCounter to int might lead to information loss and application misbehavior. You're right, so we might change the original contract and throw a UnsupportedOperationException - which is a RuntimeException, therefore the exception environment rule is not enforced, even the extended signature remains the same. Well, almost, but that's another story, so just let it go this time).

> *Comment: If you know what I mean under the term "contract", you're safe to skip this paragraph.*
> *The term "contract" in this context means the specification given together with the class. That is, it describes all the details what this class is for, how it behaves, what kind of operations/services it provides, and so on. That's what one sees in the API docs, which is used by those who are going to utilize my class. Once it is published, care must be taken to preserve the backward compatibility, which allows the already existing users to upgrade to the newest release whenever they find it appropriate. In the OO world, it is allowed to do any changes in the internals of an object, as long as the contract is kept.*

So using the getters/setters this way is rather the profanation of the OO principles than following them. After all, is it indeed an information to be exposed to the public that my class uses a counter? Unlikely. In most of the cases, definitely not. Then in some other cases it is. So how could one decide whether a field should be exposed to public this way or not? The answer lies in the OO design. The real question to ask is, whether my field represents a business entity, or it is just a technical aid. In the former case, it is okay if we provide accessor and mutator method for that field, and in the later case, we should avoid that (well, never say never, so there might be cases when it has its own merit; see more about it later). Consider the next code snipplet:

Access business info, not implementation-specific one

```
class Person {
        private String familyName;
        private String middleName;
        private String givenName;

        public String getName() { return givenName + " " + middleName + " " + familyName; }
        public String setName(String name) {
                String[] parts = name.split("\\s+");
                givenName = parts[0];
                if (parts.length == 3) {
                        middleName = parts[1];
                        familyName = parts[2];
                } else if (parts.length == 2) {
                        middleName = "";
                        familyName = parts[1];
                } else {
                        // do whatever
                }
        }
```

I know - the implementation of setName should be smarter than just that. But first, it is clearer this way, and second, many programmers addressing only the US market might live with that easily. After all, we, poor Eastern-European programmers, had to live with the ASCII character coding for long, and we were stranded with our funny accented letters (and don't even mention our Asian friends, who came off even more badly for long). So just let it this way, and concentrate on the real issue instead.

In this code above, we provide access to the "name" as a business conception. No access to the "family name", "middle name", and "given name" - those are the implementation-specific parts of the conception. Suppose we have to sell the code to Indonesia, where there's a tribe in that every person has only one unique name. Suppose we have to sell the code to Philippines, where people may have four or even more official name tags. Suppose we want our application to run in Spanish or Portuguese environment, where some noble persons may have names like "Antonio Coimbra de La Coronilla y Azevedo". Our application should be reviewed fully, finding all the occurrences were we use the fact that the name consists of two or three parts. But if we use the approach above, the rest of the application knows nothing about the internal structure of the "name" business entity, so we might make the necessary changes in this "Person" class, without having to rewrite half of the application. Conclusion is simple, due to the fact that the class provides accessor and mutator methods to access and mutate business entities, and that it is definitely legal for the rest of the application to know about this business entity, therefore the usage of the accessor/mutator method is well put. Were we providing access to implementation-dependent terms (how we represent the "name" business entity internally) by such methods, we definitely would do harm.

> **Conclusion 1:** Accessor and mutator methods should allow access to business entities, not to their implementation-dependent representation.

I might word it this way too. Using the accessor and mutator methods must be a design decision, and not a mere question of convenience.

So far so good, half of the story is perhaps told. Now we know when to use these methods. The next part of the story is about the how. How do we use them properly.

Suppose you go to your well deserved holiday, and you give the key of your house to your brother, and ask him to keep an eye on it. When you return, he is waiting for you on the airport, and gives you back your key. It's harmless, you don't see a scratch on it. Yet, you'll ask him immediately whether everything is fine at home or not. The state of the key says nothing about the house of that key it is. Now suppose we have a data model, which uses a List internally to store the data, and generates events whenever new items are added to that List, or items are removed from it.

Data Model

```
class DataModel {
	private List<MyDataItem> _data = new LinkedList<MyDataItem>();

	protected void fireEvent(final DataModelChangeEvent event) {
		// notifies the registered event listeners about the change - code omitted for better
visibility.
	}

	public void addItem(final MyDataItem newItem) {
		_data.add(newItem);
		fireEvent(new DataModelChangeEvent(this, ITEM_ADDED, newItem));
	}

	public void removeItem(final MyDataItem itemToRemove) {
		_data.remove(itemToRemove);
		fireEvent(new DataModelChangeEvent(this, ITEM_REMOVED, itemToRemove));
	}

	public List<MyDataItem> getData() {
		return _data;
	}
	public void setData(final List<MyDataItem> list) {
		_data = list;
		fireEvent(new DataModelChangeEvent(this, MODEL_CHANGED, null));
	}
}
```

I'm sure now you have the point, but just allow me to emphasize it. The key is the reference returned by the getData method, and the house is the content of the list. In the model, whenever an item is added or removed, the notification is sent automatically. Then, for whatever reason (e.g. due to a misinterpreted performance enhancement), the List is exposed to the public. Yeah, the _data will refer to the very same List instance, regardless what happens in the code that calls the getData() method. But the content of the list is not safe any more. Whoever grabs a reference to the list (that's what the getData method provides), may make changes by adding or removing elements - and the model won't know anything about it. That's, ladies and gentlemen, is one of the best ways of inducing hardly discoverable, randomly occurring bugs into our application. In this particular case, the solution is simple - use the Collections class to create an immutable version of our list and provide that reference to the public. Alternatively, and if we have to allow the external code to change the content of the list, then instead of giving the reference, implement the Iterable interface, and return with a local implementation of the Iterator, which sends the proper notifications. But all these are possible only because we're talking about a collection. But the problem is more general than that - if part of the state of the object is another object (and a reference to that other object is held), then we shouldn't allow access to that other object, unless it is immutable. And how to do it in general? Sorry, no general solution exists. It's our responsibility to keep this rule.

The mutator method is also unsafe, although at first one might think it isn't. But what if the caller keeps that reference that he has passed through, and makes changes in the list in a later time? Similar problem, and the solution is also easy - instead of storing the reference, create a new List containing the very same items.

> **Conclusion 2:** Whenever the accessor returns with objects, or the setter receives one, do not allow the direct usage of that reference, unless the referred object is immutable.

Finally, since it is very rare that universally acceptable truth would exist, therefore it is worth to note that there are cases when both of the two rules above should be broken. One such case is the data object, traveling between the client and the server (or vice versa). There, it is not just well but even wished that the object be serialized upon sending, and deserialized upon receiving. There are general frameworks that use getters and setters during the serialization and deserialization. In this case it is allowed to "expose" implementation details - after all, we're not dealing with business processes right now, we have to take care of mere technical problems, namely transferring data from one computer (e.g. the server) to another (e.g. the client). So it's actually not breaking of any of the rules above. However, in these cases, extra care should be taken to avoid the misuse of these getters and setters - they are only for sending and receiving information, and not for any other purposes.