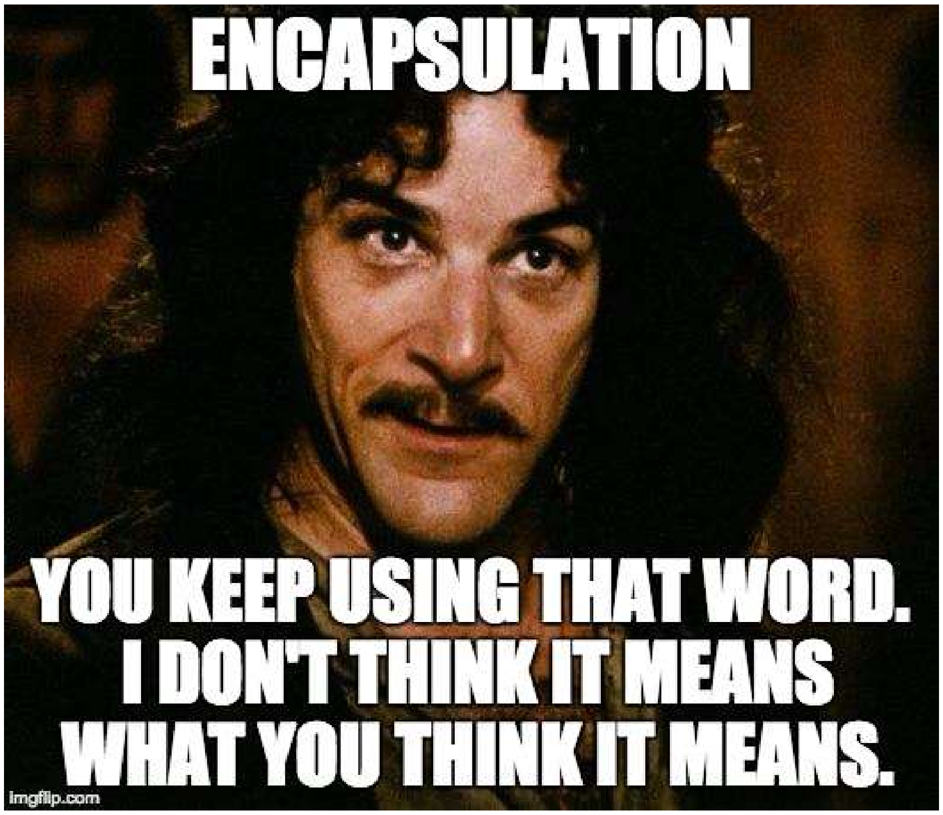
Encapsulation: I Don't Think it Means What You Think it Means

by Nicolas Frankel MVB · Jun. 07, 16 · Java Zone · Opinion



My post about immutability provoked some stir and received plenty of comments, from the daunting to the interesting, both on reddit and here.

Comment Types

They can be more or less divided into those categories:

- Let's not consider anything and don't budge an inch - with no valid argument beside "it's terrible"
- One thread wondered about the point of code review, to catch bugs or to share knowledge

- Rational counter-arguments that I'll be happy to debate in a future post
- *"It breaks encapsulation!"* This one is the point I'd like to address in this post.

Encapsulation. Really?

I've already written about encapsulation but if the wood is very hard, I guess it's natural to hammer down the nail several times.

Younger, when I learned OOP (Object-Oriented Programming), I was told about its characteristics:

1. Inheritance
2. Polymorphism
3. Encapsulation

This is the definition found on Wikipedia:

Encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

- A language mechanism for restricting direct access to some of the object's components.
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

In short, encapsulating means no direct access to an object's state but only through its methods. In Java, that directly translated to the JavaBeans conventions with private properties and public accessors - getters and setters. That is the current sad state that we are plagued with and that many refer to when talking about encapsulation.

For this kind of pattern is **no** encapsulation at all! Don't believe me? Check this snippet:

```
public class Person {

    private Date birthdate = new Date();

    public Date getBirthdate() {
        return birthdate;
    }
}
```

Given that there's no setter, it shouldn't be possible to change the date inside a `Person` instance. But it is:

```
Person person = new Person();
Date date = person.getBirthdate();
date.setTime(0L);
```

Ouch! State was not so well-encapsulated after all...

It all boils down to one tiny little difference: we want to give access to the value of the birthdate but we happily return the reference to the `birthdate` field which holds the value. Let's change that to separate the happily return the reference to the `birthdate` field which holds the value. Let's change that to separate the value itself from the reference:

```
public class Person {

    private Date birthdate = new Date();

    public Date getBirthdate() {
        return new Date(birthdate.getTime());
    }
}
```

By creating a new `Date` instance that shares nothing with the original reference, real encapsulation has been achieved. Now `getBirthdate()` is safe to call.

Note that classes that are by nature *immutable* - in Java, primitives, `String` and those that are developed like so, are completely safe to share. Thus, it's perfectly acceptable to make fields of those types `public` and forget about getters.

Note that injecting references *e.g.* in the constructor entails the exact same problem and should be treated in the same way.

```
public class Person {

    private Date birthdate;

    public Person(Date birthdate) {
        this.birthdate = new Date(birthdate.getTime());
    }

    public Date getBirthdate() {
        return new Date(birthdate.getTime());
    }
}
```

The problem is that most people who religiously invoke encapsulation blissfully share their field references to the outside world.

Conclusions

There are a couple of conclusions here:

- If you have mutable fields, simple getters such as those generated by IDEs provide **no** encapsulation.
- True encapsulation can only be achieved with mutable fields if copies of the fields are returned, and not the fields themselves.
- Once you have immutable fields, accessing them through a getter *or* having the field `final` is exactly the same.

Note: kudos to you if you understand the above *meme* reference.

Like This Article? Read More From DZone

How to Create an Immutable Class in Java

Protect Your Immutable Object Invariants in More Complex Java Objects

The Importance of Immutability in Java

Free DZone Refcard
Getting Started With Kotlin

Topics: JAVA , IMMUTABILITY