Constructor vs. Getter: A Better Way

By **Sam Atkinson**, dzone.com
March 11th, 2016

One of my jobs as a Zone Leader at DZone is to syndicate content; that is, read posts from the giant firehose of articles from our MVBs and select the best ones for republishing on DZone (aside; we're always looking for new MVBs. If you want your blog to be read by thousands of people then click this link). My interest was particularly peeked when I saw "Constructor or Setter" on the Let's Talk About Java blog. I'm always interested in the ways different people code in the hope I can learn something new, or pass on some knowledge.

Honesty from the start; I hate setters. They're generally the wrong thing to do and represent a code smell from bad wiring. Wherever possible (which is most of the time) you should pass everything in you need to the constructor and generally assign it to a final field. I don't think that's going to blow any minds as it's general good OO practice. If you're writing code using TDD you'll never inject something via a setter.

The example used in the article is an interesting one, though. Your code base is evolving and as a result, you end up with multiple constructors due to evolving optional client demands; they'd like the option of having notifications on a trigger, they'd like the option of a snapshot etc.etc. which results in the below code.

```
public class SomeHandler {
   public SomeHandler(Repository repository, Trigger
trigger) {
       // some code
   }

   public SomeHandler(Repository repository, Trigger
trigger, SnapshotTaker snapshotTaker) {
       // some code
   }

   public SomeHandler(Repository repository, Trigger
trigger, Notifier notifier) {
       // some code
   }

   public SomeHandler(Repository repository, Trigger
trigger, SnapshotTaker snapshotTaker, Notifier notifier) {
       // some code
   }

   public void handle(SomeEvent event) {
       // some code
   }
}
```

Everyone can agree this code is ugly and confusing; so many overridden constructors are far from ideal, and they're the wrong thing to do. From the original article:

"Why do we have so many constructors? Because some dependencies are optional, their presence depends on external conditions... we should place in the constructor only the required dependencies. The constructor is not a place for anything optional."

So far so good. The solution proposed is to effectively use setters, although not using "setX", but instead **enable()** methods where the optional dependency is set.

```
public class SomeHandler {
   public SomeHandler(Repository repository, Trigger
trigger) {
       // some code
   }

   public void enable(SnapshotTaker snapshotTaker) {
       // some code
   }

   public void enable(Notifier notifier) {
       // some code
   }

   public void handle(SomeEvent event) {
       // some code
   }
```

There is, however, another way which I strongly advocate people to take which I believe is cleaner in design and code, and will have fewer bugs.

The problem here is the premise that these dependencies are optional. In reality they are not; whether they are present or null, a code path will have to be called. In the example given, the code would likely look like this:

```
    public void handle(SomeEvent event) {
         Data data = repository.getData(event);
         if(snapshotTaker != null){
             snapshotTaker.snapshot(data);
         }
         if(notifier!= null){
             notifier.notify(data);
         }
         trigger.fire(data)
     }
```

Those two if statements dramatically increase the cyclomatic complexity of this method. You would need tests to make sure it functions when:

snapshotTaker and notifier are present
snapshotTaker and notifier are null
snapshotTaker is present and notifier is null
snapshotTaker is null and notifier is present

And that's before you get to testing the original functionality in the class!

In reality, these fields are not optional as there are code paths exercised. Plus, null is a terrible thing. Instead, the code should be reduced to the single constructor which takes all three fields. **If the instantiating class doesn't need the "optional" functionality then it owns the responsibility of creating and passing in a no-op version.**

In the examples, let's assume notifier and snapshotTaker only have one method, notify and snapshot respectively. This makes this really easy in Java 8:

```
new SomeHandler(new DBRepository(), new EmailTrigger(), data
->{}, data -> {});
```

Alternatively, we can have specific no-op classes for clarities sake.

```
new SomeHandler(new DBRepository(), new EmailTrigger(), new
NoOpSnapshotTaker(), new NoOpNotifier());
```

As a result, there are no if statements in the executing code, making it clearer to read and simpler to test, and less error prone.

The case of optional subscribers is particularly interesting, though. Often these may not be known at construction time, in which case it makes sense to have a specific subscribe (Subscriber sub) method. Often though it makes sense to allow multiple subscribers so these can be stored in a List. Again, if there's zero subscribers it's no big deal as the List is owned by the class and will never be null.

If we apply this to the case of the notifier in the previous example:

```
private final List<Notifier> notifiers = new LinkedList<>();

  public SomeHandler(Repository repository, Trigger trigger,
SnapshotTaker snapshotTaker) {
        // some code
        this.repository = repository;
        this.trigger = trigger;
        this.snapshotTaker = snapshotTaker;
    }

        public void handle(SomeEvent event) {
        Data data = repository.getData(event);
        snapshotTaker.snapshot(data);
        for (Notifier notifier : notifiers) {
            notifier.execute(data);
        }
        trigger.fire(data);
    }
```

Clean code with zero complexity. I Hope this offers a compelling alternative in the battle between constructors and setters.