Getters and Setters are Evil

Introduction

Since I started programming in Java in 2001, I have read so much about encapsulation. The Internet has become replete with Java articles and lessons teaching the basics of encapsulation in the form of declaring some `class` with `private` fields and `public` getters and setters.

Typical Examples of Encapsulation

Many Java books would show something like this to demonstrate encapsulation:

```
public class Car {
     private double gallonsOfFuel;
     private double milesPerGallon;

     public void setGallonsOfFuel(double gallonsOfFuel) {
          this.gallonsOfFuel = gallonsOfFuel;
     }
     public double getGallonsOfFuel() {
          return gallonssOfFuel;
     }
     public void setMilesPerGallon(double milesPerGallon) {
          this.milesPerGallon = milesPerGallon;
     }
     public double getMilesPerGallon() {
          return gallonsOfFuel;
     }
}
```

Example 1

As many such introductory articles posit, since `gallonsOfFuel` is `private`, you need to add some additional logic like decreasing `gallonsOfFuel` by some amount every time `getGallonsOfFuel()` is called. This would then be considered a well-encapsulated `Car` class, i.e., the client would have to minimally change any code in other classes if functionality in the `Car` class was changed. However, this is not the case. Take a look at the `travelMaxDistance()` function below. Just how much better is the example with getters and setters (Example 2) than one using `public` fields (Example 3)?

```
public void travelMaxDistance(Car car) {
  double maxDistance = car.getLitersOfFuel() * car.getMilesPerLiter();
  car.setLitersOfFuel(0);
  System.out.println("Car has traveled a distance of " + maxDistance + "
miles.");
}
```

```
public class CarService {
     public void travelMaxDistance(Car car) {
       System.out.println("Car traveled " + maxDistance(car) + " miles.");
     }
     private double maxDistance(Car car) {
          return car.getGallonsOfFuel() * car.getMilesPerGallon();
     }
}
```

Example 2

```
public class CarService {
     public void travelMaxDistance(Car car) {
       System.out.println("Car traveled " + maxDistance(car) + " miles.");
     }
     private double maxDistance(Car car) {
          return car.gallonsOfFuel * car.milesPerGallon;
     }
}
```

Example 3

As you can hopefully see, it isn't any better at all! If you are trying to code in an object-oriented manner, as you should when using the Java language, you shouldn't be violating encapsulation. Both of these examples violate encapsulation heavily and Example 2 represents what Martin Fowler calls an Anemic Domain Model.

Getters and Setters Are Evil

In my opinion, modern development environments have made the misuse of getters and setters all too easy by introducing automated getter and setter generation. When people start learning programming, they often find themselves introduced to auto-generated getters and setters in their IDE. One click, and people think their `class` is done and follows encapsulation rules when in fact **it couldn't be any farther from the truth.** Just because you have the power to create getters and setters with a single click, doesn't mean you're wielding your power wisely.

In cases where you are modelling some complex business behavior between objects, try to avoid automated code generation, especially getter and setter generation, as much as possible. Design your entities by adding `public` attributes only when needed. View *your entities from the perspective of its clients*. Methodologies like test-driven development easily allow the design of classes with a focus on how a client can use the code - with tests driving and verifying the design. The misuse of automatic getter and setter generation is often a symptom of a poor design - often resulting in monolithic, unmanageable "spaghetti code".

Proper Encapsulation

What methods would a client of your `class` need? Would a client be interested in the relationship between `fuel` and `miles` per gallon? If so, design for it. Make it as easy as possible for a possible client of your entity. Provide the fewest possible public methods your client needs. Do the work for your client internally. **Don't force your client to get *his hands dirty***. A real-world analogy would be a counter at a restaurant. You as a customer shouldn't need to (and in many cases aren't allowed to) know how your order is handled, only that the food comes out right. This is, roughly speaking, how encapsulation is supposed to play out. If a client was getting his hands dirty and preparing the meal himself, what would be the point of the restaurant? Think of your code in the same way.

As much as possible, try not to return raw data as directly stored in your entities' `private` fields. Always try to add some value for your clients. As an example, given that calling `travelMaxDistance()` uses all of our `Car` class' `private` fields, why not move the logic to the `Car` class so that clients would not need to know how our `Car` stores fuel and miles per gallon data? Doing so would have the net effect of reducing the amount of `Car's` public methods, which would make it easier to make changes to the car class. A smaller, more meaningful number of public methods is always preferable to multiple public methods. To this end, we could move our calculation of the maximum travel distance to our `Car` class itself:

```
public class Car {
     private double gallonsOfFuel;
     private double milesPerGallon;

     public Car(double gallonsOfFuel, double milesPerGallon) {
          this.gallonsOfFuel = gallonsOfFuel;
          this.milesPerGallon = milesPerGallon;
     }

     public double maxDistance() {
          return gallonsOfFuel * milesPerGallon;
     }
}
```
Example 4

We could then easily rewrite our `CarService` class's previous `travelMaxDistance()` method as follows:

```
public void travelMaxDistance(Car car) {
     System.out.println("Car traveled " + car.maxDistance() + " miles.");
}
```
Example 5

Note that the code now reads better as it clearly speaks the language of the business and solves their problem more precisely.

If you go back and look at the original code in Example 1 and Example 2 it is much less precise. The `Car` class was cluttered with simple but *unnecessary* code that has to be understood and maintained for no reason. You might question my changes in Example 4 since you might not think they are necessary for such a small code sample, but even there it makes our code more clear and precise and lets us see the actual important aspects of the code more easily.

As a final note, I'd like to highlight that we didn't add the "get" prefix to our `maxDistance` identifier as it decreases the readability of our code and shifts the code from an Object Oriented Design to a more procedural machine-like model.

These tiny changes, when applied rigorously to all your entities, will radically improve the readability and usability of your code. For an extensive guide that extends many of the points that a briefly talked about in this article, I would suggest reading up on Eric Evans' *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

Thanks for reading!