Chapter 3: Getters and Setters

This article discusses getters and setters; the standard way to provide access to data in Java classes.

Section 3.1: Using a setter or getter to implement a constraint

Setters and Getters allow for an object to contain private variables which can be accessed and changed with restrictions. For example,

```
public class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if(name!=null && name.length()>2)
           this.name = name;
    }
}
```

In this `Person` class, there is a single variable: `name`. This variable can be accessed using the `getName()` method and changed using the `setName(String)` method, however, setting a name requires the new name to have a length greater than 2 characters and to not be null. Using a setter method rather than making the variable `name` public allows others to set the value of `name` with certain restrictions. The same can be applied to the getter method:

```
public String getName(){
   if(name.length()>16)
      return "Name is too large!";
   else
      return name;
}
```

In the modified `getName()` method above, the `name` is returned only if its length is less than or equal to 16. Otherwise, `"Name is too large"` is returned. This allows the programmer to create variables that are reachable and modifiable however they wish, preventing client classes from editing the variables unwantedly.

Section 3.2: Why Use Getters and Setters?

Consider a basic class containing an object with getters and setters in Java:

```
public class CountHolder {
  private int count = 0;

  public int getCount() { return count; }
  public void setCount(int c) { count = c; }
}
```

We can't access the `count` variable because it's private. But we can access the `getCount()` and the `setCount(int)` methods because they are public. To some, this might raise the question; why introduce the middleman? Why not just simply make they count public?

```
public class CountHolder {
  public int count = 0;
}
```

For all intents and purposes, these two are exactly the same, functionality-wise. The difference between them is the extensibility. Consider what each class says:

- **First**: "I have a method that will give you an `int` value, and a method that will set that value to another `int`".
- **Second**: "I have an `int` that you can set and get as you please."

These might sound similar, but the first is actually much more guarded in its nature; it only lets you interact with its internal nature as **it** dictates. This leaves the ball in its court; it gets to choose how the internal interactions occur. The second has exposed its internal implementation externally, and is now not only prone to external users, but, in the case of an API, **committed** to maintaining that implementation (or otherwise releasing a non-backward-compatible API).

Lets consider if we want to synchronize access to modifying and accessing the count. In the first, this is simple:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

but in the second example, this is now nearly impossible without going through and modifying each place where the `count` variable is referenced. Worse still, if this is an item that you're providing in a library to be consumed by others, you do **not** have a way of performing that modification, and are forced to make the hard choice mentioned above.

So it begs the question; are public variables ever a good thing (or, at least, not evil)?

I'm unsure. On one hand, you can see examples of public variables that have stood the test of time (IE: the `out` variable referenced in `System.out`). On the other, providing a public variable gives no benefit outside of extremely minimal overhead and potential reduction in wordiness. My guideline here would be that, if you're planning on making a variable public, you should judge it against these criteria with **extreme** prejudice:

1. The variable should have no conceivable reason to **ever** change in its implementation. This is something that's extremely easy to screw up (and, even if you do get it right, requirements can change), which is why getters/setters are the common approach. If you're going to have a public variable, this really needs to be thought through, especially if released in a library/framework/API.
2. The variable needs to be referenced frequently enough that the minimal gains from reducing verbosity warrants it. I don't even think the overhead for using a method versus directly referencing should be considered here. It's far too negligible for what I'd conservatively estimate to be 99.9% of applications.

There's probably more than I haven't considered off the top of my head. If you're ever in doubt, always use getters/setters.

Section 3.3: Adding Getters and Setters

Encapsulation is a basic concept in OOP. It is about wrapping data and code as a single unit. In this case, it is a good practice to declare the variables as `private` and then access them through `Getters` and `Setters` to view and/or modify them.

```
public class Sample {
  private String  name;
  private int age;

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

These private variables cannot be accessed directly from outside the class. Hence they are protected from unauthorized access. But if you want to view or modify them, you can use Getters and Setters.

`getXxx()` method will return the current value of the variable `xxx`, while you can set the value of the variable `xxx` using `setXxx()`.

The naming convention of the methods are (in example variable is called `variableName`):

- All non **`boolean`** variables

```
getVariableName()   //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase
```

- **`boolean`** variables

```
 isVariableName()     //Getter, The variable name should start with uppercase
 setVariableName(...) //Setter, The variable name should start with uppercase
```

Public Getters and Setters are part of the Property definition of a Java Bean.