Setters and Getters in Java

In Java, getter and setter are two **conventional methods** that are used for **retrieving and updating value of a variable**.

As we **cannot access** the **private members** of the class **directly outside** the class. But we can give **protected access** by providing setters and getters.

So, a **setter** is a method that **updates value of a variable**. And a **getter** is a method that **reads value of a variable**.

Getter and setter are also known as ***accessor*** and ***mutator*** in Java.

Why setter and getter?

So far, setter and getter methods protect a variable's value from unexpected changes by outside world - the caller code.

When a variable is hidden by *private* modifier and can be accessed only through getter and setter, it is ***encapsulated***. Encapsulation is one of the **fundamental principles** in object-oriented programming (OOP), thus implementing **getter and setter is one of ways to enforce encapsulation** in program's code.

Some frameworks such as Hiberate, Spring, Struts… can inspect information or inject their utility code through getter and setter. So providing getter and setter is necessary when integrating your code with such frameworks.

Naming Convention for getter and setter

The naming scheme of setter and getter should follow **Java bean naming convention** as follows:

getXXX() and setXXX()

where XXX is **name of the variable**. For example with the following variable name:

private String name;

Then the appropriate setter and getter will be:

public void setName(Strinf name) { }

public String getName() { }

If the variable is of type **boolean**, then the getter's name can be either **isXXX() or getXXX()**, but the former naming is preferred.

For example:

private Boolean single;

public String isSingle( ) { }

Common mistakes while implementing getters and setters

People often make mistakes, so do developers. This section describes the most common mistakes when implementing setter and getter in Java, and workarounds.

Mistake #1: Have setter and getter, but the variable is declared in less restricted scope.

If the variable is declared as public, then it can be accessed using dot (.) operator directly, making the setter and getter useless. Workaround for this case is using more restricted access modifier such as protected and private.

Mistake #2: Assign object reference directly in setter

Considering the following setter method:

```
private int[] scores;

public void setScores(int[] scr) {
    this.scores = scr;
}
```

And following is code that demonstrates the problem:

```
int[] myScores = {5, 5, 4, 3, 2, 4};

setScores(myScores);
displayScores();
myScores[1] = 1;
displayScores();
```

An array of integer numbers myScores is initialized with 6 values (line 1) and the array is passed to the setScores() method (line 2). The method displayScores() simply prints out all scores from the array:

```
public void displayScores() {
    for (int i = 0; i < this.scores.length; i++) {
        System.out.print(this.scores[i] + " ");
    }
    System.out.println();
}
```

Line 3 will produce the following output:

5 5 4 3 2 4

That are exactly all elements of the myScores array.
Now at line 4, we can modify the value of the 2nd element in the myScores array as follow:

myScores[1] = 1;

What will happen if we call the method displayScores() again at line 5? Well, it will produce the following output:

5 1 4 3 2 4

You can realize that the value of 2nd element is changed from 5 to 1, as a result of the assignment in line 4. Why does it matter? Well, that means the data can be modified outside scope of the setter method which breaks encapsulation purpose of the setter. And why that happens? Let's look at the setScores() method again:

```
public void setScores(int[] scr) {
    this.scores = scr;
}
```

The member variable scores is assigned to the method's parameter variable scr directly. That means both the variables are referring the same object in memory - the myScores array object. So changes made to either scores variable or myScores variable are actually made on the same object.
Workaround for this situation is to copy elements from scr array to scores array, one by one. The modified version of the setter would be like this:

```
public void setScores(int[] scr) {
    this.scores = new int[scr.length];
    System.arraycopy(scr, 0, this.scores, 0, scr.length);
}
```

What's difference? Well, the member variable scores is no longer referring to the object referred by scr variable. Instead, the array scores is initialized to a new one with size equals to the size of the array scr. Then we copy all elements from the array scr to the array scores, using System.arraycopy() method.

Run the example again and it will give the following output:

```
5  5  4  3  2  4
5  5  4  3  2  4
```

Now the two invocation of displayScores()produce the same output. That means the array scores is independent and different than the array scr passed into the setter, thus the assignment:

myScores[1] = 1;

does not affect the array scores.
So, the rule of thumb is, if you pass an object reference into a setter method, then don't copy that reference into the internal variable directly. Instead, you should find some ways to copy values of the passed object into the internal object, like we have copied elements from one array to another using System.arraycopy() method.

Mistake #3: Return object reference directly in getter

Consider the following getter method:

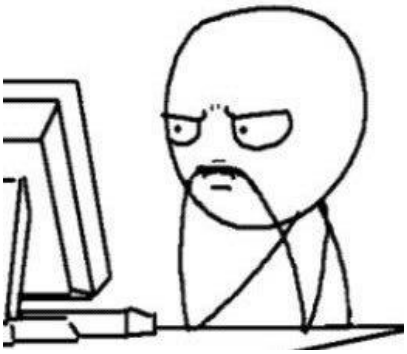
```
private int[] scores;

public int[] getScores() {
    return this.scores;
}
```

And the following code snippet:

```
int[] myScores = {5, 5, 4, 3, 2, 4};

setScores(myScores);
displayScores();
int[] copyScores = getScores();
copyScores[1] = 1;
displayScores();
```



it will produce the following output:

```
5  5  4  3  2  4
5  1  4  3  2  4
```

As you notice, the 2nd element of the array scores is modified outside the setter, at line 5. Because the getter method returns reference of the internal variable scores directly, so the outside code can obtain this reference and makes change to the internal object.

Workaround for this case is that, instead of returning the reference directly in the getter, we should return a copy of the object, so the outside code can obtain only a copy, not the internal object. Therefore we modify the above getter as follows:

```java
public int[] getScores() {
    int[] copy = new int[this.scores.length];
    System.arraycopy(this.scores, 0, copy, 0, copy.length);
    return copy;
}
```

So the rule of thumb is: do not return reference of the original object in getter method. Instead, it should return a copy of the original object.

Variable Shadowing

If the **instance variable and local variable** have **same name** whenever you print (access) it in the method. The **value of the local variable will be printed** (shadowing the instance variable).

If you still, need to access the **values of instance variables** in a method you need to access them using **this keyword** (or object).

Now let's see the special type of setter which are called as Constructors

Constructors are special type of setters which follow the following rules:

- Method name will be same as that of the class name.
- No return type.
- Arguments are passed during the object creation.

Note: Whenever the programmer **does not provide a single constructor**, then during the **object creation** when we call the constructor, we are basically calling the **default constructor**.

Is there any difference between constructor and method???

Well definitely there is a difference between method and constructor. So let us see what the difference is…

Sl. No.	Constructor	Method
1	Constructor name should be same as class name.	Method name may or may not be same as class name.
2	Constructor never returns any value so it has no return type.	Method must have a return type in Java and returns only a single value.
3	There are 2 types of constructor available in Java.	Java supports 6 types of method.
4	Constructor only can be called by new keyword in Java	Method can be called by class name, object name or directly.
5	If there is no constructor designed by user then the Java compiler automatically provides the default constructor.	Javac never provides any method by default.
6	Constructor cannot be overridden in Java.	Method can be overridden.
7	Constructor cannot be declared as static.	Method can be declared as Static.