Getter and Setter Method in Java

Getter and setter methods are frequently used in Java programming. **Getter and setter methods in Java** are widely used to access and manipulate the values of class fields. Usually, class fields are decorated with a private access specifier. Thus, to access them, public access specifiers are used with the getter and setter methods.

The Need of Getter and Setter Method

One may argue that declare the class fields as public and remove the getter and setter methods. However, such a coding style is bad, and one may put some absurd value on the class fields. Let's understand it with the help of an example.

1. **public** **class** GetterSetterExample
2. {
3. **public** salary;
5. **public** storeSalaryDB(**int** salary)
6. {
7. // code for storing the salary in the database
8. }
10. // main method
11. **public** **static** **void** main(String argvs[])
12. {
13. GetterSetterExample obj = **new** GetterSetterExample();
15. obj.salary = -50000;
17. // storing salary in database
18. obj.storeSalaryDB(salary);
20. }
21. }

Observe that the code is storing a negative salary in the database that is wrong. An organization never credits a negative salary to the account of an employee. Assigning an absurd amount to the salary variable happened because it is declared with a public access specifier. The correct way to write the above code is:

1. **public** **class** GetterSetterExample
2. {
3. **private** salary;
5. // a setter method that assign a
6. // value to the salary variable
7. **void** setSalary(**int** s)
8. {
9. **if**(s < 0 )
10. {
11. s = -s;
12. }
14. **this**.salary = s;
15. }

18. // a getter mehtod to retrieve
19. // the salary
20. **int** getSalary()
21. {
22. **return** **this**.salary;
23. }
25. **public** storeSalaryDB(**int** salary)
26. {
27. // code for storing the salary in the database
28. System.out.println("The ")
29. }
31. // main method
32. **public** **static** **void** main(String argvs[])
33. {
34. // creating an object of the class GetterSetterExample
35. GetterSetterExample obj = **new** GetterSetterExample();
37. obj.setSalary(-50000);
39. **int** salary = obj.getSalary();
41. // storing salary in database
42. obj.storeSalaryDB(salary);
44. }
45. }

Now, we can see better control over what we send to the database to store. Whenever the salary is negative, we are converting the salary into a positive value, and then we are sending it to the database to store. Thus, no matter what value we send to the setter method, the if-block of the setter method takes care of the absurd value and thus gives better control on the salary value.

Getter Setter Java Program

**FileName:** GetterSetterExample1.java

1. **class** Employee
2. {
3. // class member variable
4. **private** **int** eId;
5. **private** String eName;
6. **private** String eDesignation;
7. **private** String eCompany;
9. **public** **int** getEmpId()
10. {
11. **return** eId;
12. }
13. **public** **void** setEmpId(**final** **int** eId)
14. {
15. **this**.eId = eId;
16. }
17. **public** String getEmpName()
18. {
19. **return** eName;
20. }
21. **public** **void** setEmpName(**final** String eName)
22. {
23. // Validating the employee's name and
24. // throwing an exception if the name is null or its length is less than or equal to 0.
25. **if**(eName == **null** ||  eName.length() <= 0)
26. {
27. **throw** **new** IllegalArgumentException();
28. }
29. **this**.eName = eName;
30. }
31. **public** String getEmpDesignation()
32. {
33. **return** eDesignation;
34. }
35. **public** **void** setEmpDesignation(**final** String eDesignation)
36. {
37. **this**.eDesignation = eDesignation;
38. }
39. **public** String getEmpCompany()
40. {
41. **return** eCompany;
42. }
44. **public** **void** setEmpCompany(**final** String eCompany)
45. {
46. **this**.eCompany = eCompany;
47. }
48. // for printing the values
49. @Override
50. **public** String toString()
51. {
52. String str = "Employee: [id = " + getEmpId() + ", name = " + getEmpName() + ", designation = " + getEmpDesignation() + ", company = " + getEmpCompany() + "]";
53. **return** str;
54. }
55. }
56. // Main class.
57. **public** **class** GetterSetterExample1
58. {
59. // main method
60. **public** **static** **void** main(String argvs[])
61. {
62. // Creating an object of the Employee class
63. **final** Employee emp = **new** Employee();
65. // the employee details are getting set using the setter methods.
66. emp.setEmpId(107);
67. emp.setEmpName("Kathy");
68. emp.setEmpDesignation("Software Tester");
69. emp.setEmpCompany("XYZ Corporation");
71. // Displaying the details of the employee details using the
72. // 'toString()' method, which uses the getter methods
73. System.out.println(emp.toString());
74. }
75. }

**Output:**

Employee: [id = 107, name = Kathy, designation = Software Tester, company = XYZ Corporation]

Bad Practices in Getter and Setter Methods

There are some common bad practices that people usually do when they deal with the getter and setter methods.

Bad Practice 1:

Using getter and setter for the variable that is declared with low restricted scope.

1. **public** salary;
3. **void** setSalary(**int** s)
4. {
5. salary = s;
6. }
8. **int** getSalary()
9. {
10. **return** salary;
11. }

It is evident that from the main method, one can directly access the variable salary, which is not only bad but also makes the presence of the getter and setter methods irrelevant.

Bad Practice 2:

Using an object reference in the setter method. Consider the following program.

**FileName:** GetterSetterExample2.java

1. **class** ABC
2. {
3. **private** **int**[] val;
5. **void** setVal(**int**[] arr)
6. {
7. **this**.val = arr; // line 7
8. }
10. // for displaying the value
11. // present in the val array
12. **void** display()
13. {
14. **int** size = (**this**.val).length;
16. **for**(**int** i = 0; i < size; i++)
17. {
18. System.out.print(**this**.val[i] + " ");
19. }
21. }
23. }
25. // Main class
26. **public** **class** GetterSetterExample2
27. {
28. // main method
29. **public** **static** **void** main(String argvs[])
30. {
31. // instantiating the class ABC
32. ABC obj = **new** ABC();
34. **int** mainArr[] = {3, 4, 6, 8, 78, 9};
36. // invoking the setter method
37. obj.setVal(mainArr);
39. // invoking the display method
40. obj.display();
42. // updating the value at the 0th index
43. mainArr[0] = -1;
45. System.out.println();
47. obj.display();
48. }
49. }

**Output:**

3 4 6 8 78 9

-1 4 6 8 78 9

**Explanation:**

References are a bit tricky to deal with! In the above code, at line 43, the value got updated at the 0th index for array mainArr[]. However, it also got reflected in the array val[]. It should not happen as val[] array is declared private; hence, it is expected that any code outside of the class ABC should not modify it. However, because of the references, everything is messed up. The setter method setVal() expecting a reference of an int array, and at line 7, the reference of the int arr[] is getting copied to val[]. Note that the reference variable arr[] is storing the reference of the array mainArr[]. Thus, we can say val[] is storing the reference of the mainArr[].

Therefore, whatever we change in the mainArr[] also gets reflected in the val[] array, which violates the purpose of the setter method. Also, there is no meaning in adding the private access specifier to the val[] array; because one can change the value of the val[] array in the main method, which is evident by looking at the output.

A better way of writing the above code is:

**FileName:** GetterSetterExample3.java

1. **class** ABC
2. {
3. **private** **int**[] val;
5. **void** setVal(**int**[] arr)
6. {
7. **int** size = arr.length;
9. // allocating the memory as
10. // per the array arr size
11. val = **new** **int**[size]; // line 11
13. **for**(**int** i = 0; i < size; i++)
14. {
15. // copying the value one by one
16. // into the val array
17. **this**.val[i] = arr[i]; // line 17
18. }
19. }
21. // for displaying the value
22. // present in the val array
23. **void** display()
24. {
25. **int** size = (**this**.val).length;
27. **for**(**int** i = 0; i < size; i++)
28. {
29. System.out.print(**this**.val[i] + " ");
30. }
32. }
34. }
35. // Main class.
36. **public** **class** GetterSetterExample3
37. {
38. // main method
39. **public** **static** **void** main(String argvs[])
40. {
41. // instantiating the class ABC
42. ABC obj = **new** ABC();
44. **int** mainArr[] = {3, 4, 6, 8, 78, 9};
46. // invoking the setter method
47. obj.setVal(mainArr);
49. // invoking the display method
50. obj.display();
52. // updating the value at the 0th index
53. mainArr[0] = -1; // line 53
55. System.out.println();
57. // invoking the display method again
58. obj.display();
60. }
61. }

**Output:**

3 4 6 8 78 9

3 4 6 8 78 9

**Explanation:**

In the above code, we are doing the deep copy of elements of the array arr[]. In line 11, we are creating an entirely new array. Thus, the val[] is not referring to the arr[]. Also, in line 17, only values of the element are getting copied. Therefore, when we change the value of the 0th element at line 53, the change is not reflected in the val[]. Thus, the above code respects the encapsulation of the private member variable val[].

Bad Practice 3:

Returning an object reference in the getter method. Observe the following program.

**FileName:** GetterSetterExample4.java

1. **class** ABC
2. {
3. **private** **int**[] val = {67, 43, 68, 112, 70, 12};
5. // the getter method
6. **public** **int**[] getVal()
7. {
8. // returning the reference
9. **return** val; // line 9
10. }
12. // for displaying the value
13. // present in the val array
14. **void** display()
15. {
16. **int** size = (**this**.val).length;
18. **for**(**int** i = 0; i < size; i++)
19. {
20. System.out.print(**this**.val[i] + " ");
21. }
23. }
25. }
26. // Main class.
27. **public** **class** GetterSetterExample4
28. {
29. // main method
30. **public** **static** **void** main(String argvs[])
31. {
32. // instantiating the class ABC
33. ABC obj = **new** ABC();
35. // invoking the getter method
36. // and storing the result
37. **int** arr[] = obj.getVal();
39. // invoking the display method
40. obj.display();
42. // updating the value at the 0th index
43. arr[0] = -1; // line 42
45. System.out.println();
47. // invoking the display method again
48. obj.display();
50. }
51. }

**Output:**

67 43 68 112 70 12

-1 43 68 112 70 12

**Explanation:**

The above code is not handling the references properly. The getter method is returning the reference of the array. The arr[] is storing the reference of the array val[], which is declared private in the class ABC. Because of exposing the reference to the outer world, arr[] can manipulate the val[], and thus, the encapsulation of the class ABC is breached. The proper way to handle the above is:

**FileName:** GetterSetterExample5.java

1. **class** ABC
2. {
3. **private** **int**[] val = {67, 43, 68, 112, 70, 12};
5. // the getter method
6. **public** **int**[] getVal()
7. {
9. **int** size = val.length;
11. // creating a new array
12. **int** temp[] = **new** **int**[size];
14. // copying the content of the array to temp array
15. **for**(**int** i = 0; i < size; i++)
16. {
17. temp[i] = val[i];
18. }
20. **return** temp;
21. }
23. // for displaying the value
24. // present in the val array
25. **void** display()
26. {
27. **int** size = (**this**.val).length;
29. **for**(**int** i = 0; i < size; i++)
30. {
31. System.out.print(**this**.val[i] + " ");
32. }
34. }
36. }
37. // Main class.
38. **public** **class** GetterSetterExample5
39. {
40. // main method
41. **public** **static** **void** main(String argvs[])
42. {
43. // instantiating the class ABC
44. ABC obj = **new** ABC();
46. // invoking the getter method
47. // and storing the result
48. **int** arr[] = obj.getVal();
49. // invoking the display method
50. obj.display();
51. // updating the value at the 0th index
52. arr[0] = -1; // line 54
53. System.out.println();
54. obj.display();
55. }
56. }

**Output:**

67 43 68 112 70 12

67 43 68 112 70 12

**Explanation:** In the above code, the reference of the private array is not sent to the outside world. In the getter method, a new array is created whose reference is sent to the main method. Therefore, when the value at the 0th index gets changed at line 54, that change impacts the temp[] array, not the private array val[]. Thus, the encapsulation of the class ABC is maintained, as the reference of the array val[] is not exposed to the outside world.

Note 1: For primitive data types (int, char, etc.), one does not need to create a copy in the getter and setter methods, as the concept of references is absent for the primitive data types.

Note 2: Strings object types also work on the references. However, unlike the above examples, one does not need to take care of the String references exposed to the outside world. It is because Strings are immutable. Thus, when one manipulates the string in the main method (or anywhere else), a new String object is created, and the previous one remains untouched.

**FileName:** GetterSetterExample6.java

1. **class** ABC
2. {
3. **private** String str = **null**;
5. // a setter method
6. **void** setVal(String s)
7. {
8. // reference is getting copied
9. **this**.str = s;
10. }
12. // for displaying the string
13. **void** display()
14. {
15. System.out.println( "The String is: " + **this**.str);
16. }
17. }
19. // Main class.
20. **public** **class** GetterSetterExample6
21. {
22. // main method
23. **public** **static** **void** main(String argvs[])
24. {
25. // creating an object of the class ABC
26. ABC obj = **new** ABC();

29. // input string
30. String inputStr = "Hello India!";
32. // invoking the setter method
33. obj.setVal(inputStr);
35. obj.display();
37. // manipulation is not allowed!
38. // it leads to the creation of the new string
39. inputStr = "Hello World!";
41. obj.display();
43. }
44. }

**Output:**

The String is: Hello India!

The String is: Hello India!