[[Manoj Chemate](https://medium.com/@manojchemate?source=post_page-----c54eac99cd3--------------------------------)](https://medium.com/@manojchemate?source=post_page-----c54eac99cd3--------------------------------)

[Manoj Chemate](https://medium.com/@manojchemate?source=post_page-----c54eac99cd3--------------------------------)

Oct 2, 2022

·

6 min read

·

Listen

**7 Mistakes That Novice Java Developers Commit!**

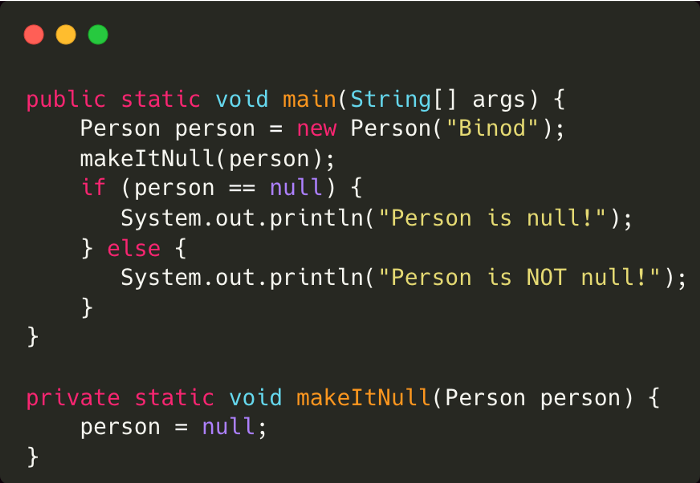
*If you are new to Java this article will help you to understand 7 confusing concepts.*



[Photo by Andrea Piacquadio](https://www.pexels.com/photo/young-annoyed-female-freelancer-using-laptop-at-home-3808008/)

**1. Ignoring P*ass By Value* Nature Of Java**

*do you see the bug here?*



Here, *person* will not be *null* since Java is *pass by value* and not *pass by reference*. what does that mean?

There are three parts here, actual person object is created on the heap, then there is a reference that points to that person object on the heap, and the reference itself has a value that represents the location of the object on the heap, here *bob* is a reference that is pointing to its person object on the heap and *bob* as a reference also has some value representing object location. ( It is not the exact address )

so,

When the *bob* is passed as a parameter to *makeItNull()*method, the value of bob reference is copied to *bob1* reference, and not the actual heap object, when *bob1* is assigned to *null*, the value of *bob1* reference becomes *null* but the value of the original *bob* reference does not change.

When passing primitive data, JVM copies the value of the primitive data type. If int x=5, then 5 itself will be copied.

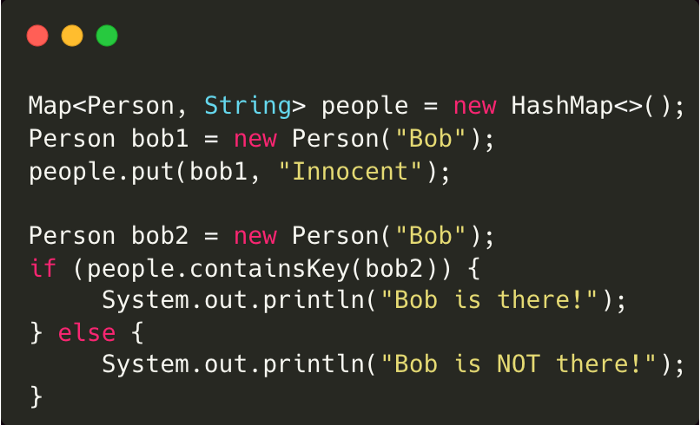
When passing object, value of reference is copied to another variable and not the actual object.

and hence java is considered *pass-by-value* and not *pass-by-reference.*

In pass-by-reference languages, the original *bob* would have become null.

**2. Breaking Hashcode-Equals Contract**

what is missing here ?



If you don’t override the hashcode-equals method in the Person class, **Bob will not be there**, even if they are the same objects from a business perspective.

why?

HashMap, HashSet, and Hashtable identify duplicate keys using hashcode and equals method,

While storing objects, the hash code integer is calculated using the *hashcode* method, if the map already contains that hash code then the equals method will be called to check if objects are really same, and if they are the same, then values will be replaced.

Similarly while retrieving objects, if the *hashcode()* is the same then *equals()* is called to check if it’s the same key and if yes then the value is returned.

Here, while storing objects, since the *hashcode()* and *equals()* method is not overridden, Object class’s hashcode and equals method will be called

Object class’s hashcode method returns a different value for every new object in JVM.

Object class’s equal method does reference comparison.

so the hashcode for bob and bob1 will be different. Hash of *bob1* will not be present in the map.

**Remember the simple contract:**

If the *equals* method is overridden in some class,

1. You must override the *hashcode* method in such a way that If the equals method returns true for two objects, then the hashcode method must return the same integer for those objects.
2. If objects are not the same by the equals method, then they may or may not return the same hashcode.

Even if you have overridden the hashcode-equals, It is not good practice to use mutable objects as keys, because later if keys get modified, they cannot be retrieved since the hashcode-equals will change results due to modification.

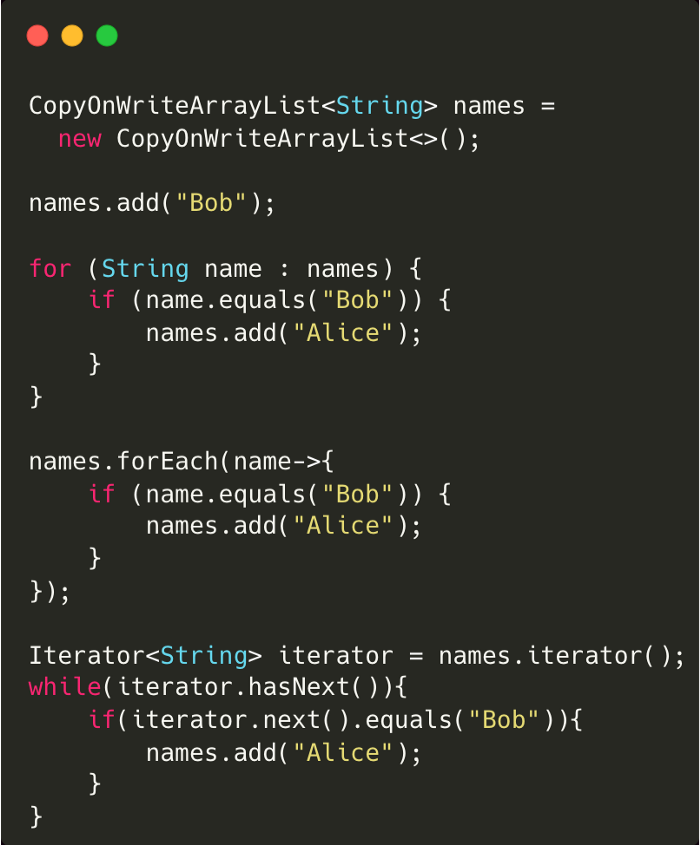
**3. Modifying Collection While Iterating**



1. If you try to add or remove objects while iterating over them using a fail-fast iterator, you will get *ConcurrentModificationException.*
2. Fail fast iterator works on the original collection and uses aninternal flag *modCount* to check if there are any structural changes in the collection. If yes, it will fail the execution.

**Solution:**

1. you can use thread-safe equivalent of collections *java.util.concurrent.CopyOnWriteArrayList*or *java.util.concurrent.CopyOnWriteArraySet* on which the iterator uses a copy of the original array to traverse and makes modifications to the original array.
2. You can also use *ConcurrentHashMap*, although it does not use a copy of the original collection to traverse, it has a non-fail fast implementation.

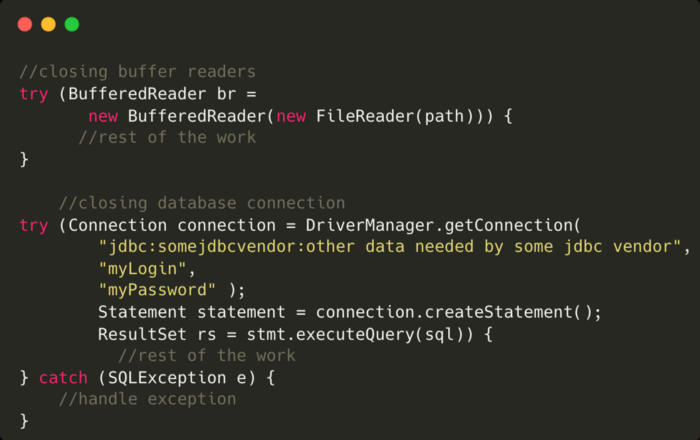


**4. Not Closing System Resources**

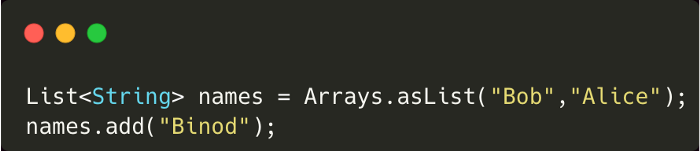
1. Java application uses several types of resources such as files, streams, sockets, and database connections. We need to ensure that they get freed **even in case of errors**.
2. Since every application has a limited of resources assigned, overusing resources leads to restarting the application multiple times and affecting other applications in the same environment.

**How to close resources?**

You can use *finally* block or *try-with-resource* block introduced in Java 7. While using *try-with-resource*, you can declare multiple resources, each resource must implement *java.io.AutoCloseable* interface which has *close()* method to specify how resources will be closed. This method will always be called by JVM after finishing the try block successfully or with an exception.



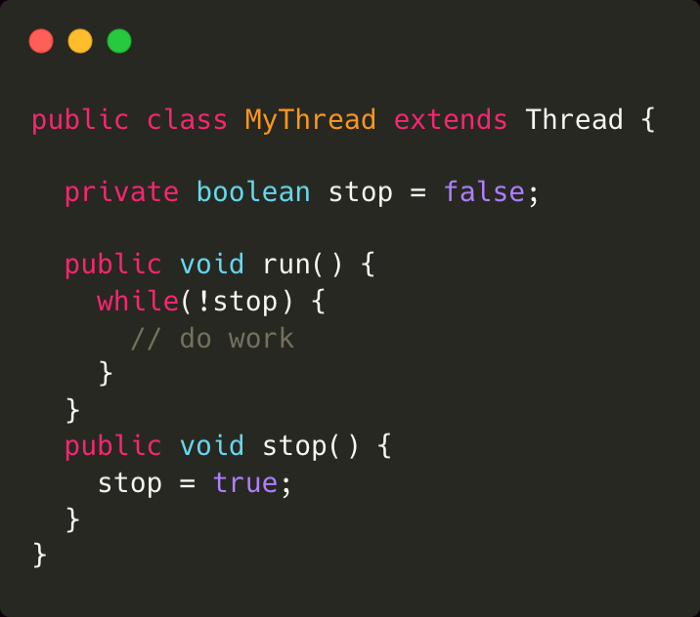
**5. Modifying Immutable Objects**



1. Here *Arrays.asList()* gives an immutable list in which we are trying to add another name hence we will get *UnsupportedOperationException*here*.*
2. The state of immutable objects can’t be changed and should not be changed. If it is required then the object itself should not be immutable.

**6. Visibility of Variable In Multithreading Environment**

Do you think MyThread will always stop here?



1. If *stop()* method is executed by some other thread, it is possible that *true* value of *stop* variable will never be visible to MyThread if it has cached variable’s value.
2. If a variable is shared by multiple threads, changes made by one thread may or may not be visible to the other thread. why?
3. When a thread tries to modify the value of a variable, depending on CPU architecture, the value of the variable may be copied to CPU core’s local cache instead of always reading it from the main memory to improve performance, which will not be visible to another thread working on it at the same time.