Why String is Immutable in Java?

1. Introduction

In Java, Strings are immutable. An obvious question that is quite prevalent in interviews is "Why Strings are designed as immutable in Java?"

James Gosling, the creator of Java, was once asked in an interview when should one use immutables, to which he answers:

> I would use an immutable whenever I can.

He further supports his argument stating features that immutability provides, such as caching, security, easy reuse without replication, etc.

In this tutorial, we'll further explore why the Java language designers decided to keep *String* immutable.

2. What Is an Immutable Object?

An immutable object is an **object whose internal state remains constant after it has been entirely created**. This means that once the object has been assigned to a variable, we can neither update the reference nor mutate the internal state by any means.

We have a separate article that discusses immutable objects in detail. For more information, read the Immutable Objects in Java article.

3. Why Is *String* Immutable in Java?

The key benefits of keeping this class as immutable are caching, security, synchronization, and performance.

Let's discuss how these things work.

3.1. Introduce to *String* Pool

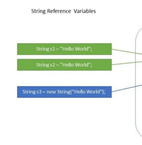
The *String* is the most widely used data structure. Caching the *String* literals and reusing them saves a lot of heap space because different *String* variables refer to the same object in the *String* pool. *String* intern pool serves exactly this purpose.

Java String Pool is **the special memory region where *Strings* are stored by the JVM**. Since *Strings* are immutable in Java, the JVM optimizes the amount of memory allocated for them by storing only one copy of each literal *String* in the pool. This process is called interning:

```
String s1 = "Hello World";
String s2 = "Hello World";

assertThat(s1 == s2).isTrue();
```

Because of the presence of the *String* pool in the preceding example, two different variables are pointing to same *String* object from the pool, thus saving crucial memory resource.



We have a separate article dedicated to Java *String* Pool. For more information, head on over to that article.

3.2. Security

The *String* is widely used in Java applications to store sensitive pieces of information like usernames, passwords, connection URLs, network connections, etc. It's also used extensively by JVM class loaders while loading classes.

Hence securing *String* class is crucial regarding the security of the whole application in general. For example, consider this simple code snippet:

```
void criticalMethod(String userName) {
    // perform security checks
    if (!isAlphaNumeric(userName)) {
        throw new SecurityException();
    }

    // do some secondary tasks
    initializeDatabase();

    // critical task
    connection.executeUpdate("UPDATE Customers SET Status = 'Active' " +
      " WHERE UserName = '" + userName + "'");
}
```

In the above code snippet, let's say that we received a *String* object from an untrustworthy source. We're doing all necessary security checks initially to check if the *String* is only alphanumeric, followed by some more operations.

Remember that our unreliable source caller method still has reference to this *userName* object.

If *Strings* were mutable, then by the time we execute the update, we can't be sure that the *String* we received, even after performing security checks, would be safe. The untrustworthy caller method still has the reference and can change the *String* between integrity checks. Thus making our query prone to SQL injections in this case. So mutable *Strings* could lead to degradation of security over time.

It could also happen that the *String userName* is visible to another thread, which could then change its value after the integrity check.

In general, immutability comes to our rescue in this case because it's easier to operate with sensitive code when values don't change because there are fewer interleavings of operations that might affect the result.

3.3. Synchronization

Being immutable automatically makes the *String* thread safe since they won't be changed when accessed from multiple threads.

Hence **immutable objects, in general, can be shared across multiple threads running simultaneously. They're also thread-safe** because if a thread changes the value, then instead of modifying the same, a new *String* would be created in the *String* pool. Hence, *Strings* are safe for multi-threading.

3.4. Hashcode Caching

Since *String* objects are abundantly used as a data structure, they are also widely used in hash implementations like *HashMap*, *HashTable*, *HashSet*, etc. When operating upon these hash implementations, *hashCode()* method is called quite frequently for bucketing.

The immutability guarantees *Strings* that their value won't change. So **the *hashCode()* method is overridden in *String* class to facilitate caching, such that the hash is calculated and cached during the first *hashCode()* call and the same value is returned ever since.**

This, in turn, improves the performance of collections that uses hash implementations when operated with *String* objects.

On the other hand, mutable *Strings* would produce two different hashcodes at the time of insertion and retrieval if contents of *String* was modified after the operation, potentially losing the value object in the *Map*.

3.5. Performance

As we saw previously, *String* pool exists because *Strings* are immutable. In turn, it enhances the performance by saving heap memory and faster access of hash implementations when operated with *Strings*.

Since *String* is the most widely used data structure, improving the performance of *String* have a considerable effect on improving the performance of the whole application in general.

4. Conclusion

Through this article, we can conclude that **Strings are immutable precisely so that their references can be treated as a normal variable and one can pass them around, between methods and across threads, without worrying about whether the actual *String* object it's pointing to will change.**

We also learned as what might be the other reasons that prompted the *Java* language designers to make this class as immutable.