# Why String is Immutable or Final in Java

In object-oriented programming, the **immutable string or objects** that cannot be modified once it is created. But we can only change the reference to the object. We restrict to change the object itself. The **String is immutable** in [Java](https://www.javatpoint.com/java-tutorial) because of the security, synchronization and concurrency, caching, and class loading. The reason of making string final is to destroy the immutability and to not allow others to extend it.

The String objects are cached in the String pool, and it makes the [String immutable](https://www.javatpoint.com/immutable-string). The cached String literals are accessed by multiple clients. So, there is always a risk, where action performs by one client affects all other clients. For example, if one client performs an action and changes the string value from Pressure to PRESSURE, all remaining clients will also read that value. For the performance reason, caching of String objects was important, so to remove that risk, we have to make the String Immutable.

1. **How to create custom immutable class in java?**

An immutable class object's properties cannot be modified after initialization. For example String is an immutable class in Java. We can create a immutable class by following the given rules below −

* **Make class final** − class should be final so that it cannot be extended.
* **Make each field final** − Each field should be final so that they cannot be modified after initialization.
* **Create getter method for each field.** − Create a public getter method for each field. fields should be private.
* **No setter method for each field.** − Do not create a public setter method for any of the field.
* **Create a parametrized constructor** − Such a constructor will be used to initialize properties once.

## What is the string pool?

**String pool** is nothing but a storage area in [Java heap](https://www.javatpoint.com/java-heap) where string literals stores. It is also known as **String Intern Pool** or **String Constant Pool**. It is just like object allocation. By default, it is empty and privately maintained by the [**Java String**](https://www.javatpoint.com/java-string) class. Whenever we create a string the string object occupies some space in the heap memory. Creating a number of strings may increase the cost and memory too which may reduce the performance also.

The JVM performs some steps during the initialization of string literals that increase the performance and decrease the memory load. To decrease the number of String objects created in the JVM the String class keeps a pool of strings.

When we create a string literal, the JVM first check that literal in the String pool. If the literal is already present in the pool, it returns a reference to the pooled instance. If the literal is not present in the pool, a new String object takes place in the String pool.

1. **Compare JDK vs JVM VS JRE.**

1. JVM

a. Virtual machine that run the Java bytecode.

b. Makes java portable.

2. JRE

a. JVM + Libraries + Other Components (to run applets and other java applications)

3. JDK

a. JRE + Compilers + Debuggers

1. **Difference between @Primary and @Qualifier annotations?**

When an interface has multiple different implementations, org.springframework.beans.factory.NoUniqueBeanDefinitionException will be generated when Spring is injected.  
 There are two solutions, using @Primary and @Qualifier annotations.

| **annotation** | **the difference** |
| --- | --- |
| @Primary | Give priority to inject the annotated bean |
| @Qualifier | Determine the bean to be injected into the annotation |

package com.zetcode.model;

public interface Person {

String info();

}

We have an interface that defines the Person type.

**com/zetcode/model/Student.java**

package com.zetcode.model;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component

@Qualifier("student")

public class Student implements Person {

@Override

public String info() {

return "Student";

}

}

Student inherits from Person. @Component is a basic Spring annotation that allows Student to be detected by Spring containter. The @Qualifier("student") uniquely identifies this bean with the "student" string.

**com/zetcode/model/Manager.java**

package com.zetcode.model;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component

@Qualifier("manager")

public class Manager implements Person {

@Override

public String info() {

return "Manager";

}

}

We have another bean called Manager. This bean is also identified with the @Qualifier("manager") annotation.

@Primary is more like the provider telling you which one to use first. For example, you implement many instances of the sort interface, such as bubble sort, fast sort, select sort and so on. In the case where the user's choice is not clear, you want him to give priority to (default) to choose bubble sorting. So you use @Primary to identify the bubble sort implementation class.

public interface SortAlgorithm {

public int[] sort(int[] numbers);

}

@Component

@Primary

public class BubbleSortAlgorithm implements SortAlgorithm {

}

1. **How hashmap get method works internally?**

The java.util.HashMap.get() method of HashMap class is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when the map contains no such mapping for the key.

Suppose we have to fetch the Key "Aman." The following method will be called.

map.get(**new** Key("Aman"));

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists.

1. **What is serialVersionUID?**

The serialization at runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

 SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded.

**Serialization :**At the time of serialization, with every object sender side JVM will save a **Unique Identifier**. JVM is responsible to generate that unique ID based on the corresponding .class file which is present in the sender system.

**Deserialization:** At the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise we will get Runtime Exception saying **InvalidClassException**. This unique Identifier is nothing but **SerialVersionUID**.

**Problem of depending on default SerialVersionUID generated by JVM :**

* Both sender and receiver should use the same JVM with respect to platform and version also. Otherwise receiver unable to deserialize because of different SerialVersionUID.
* Both sender and receiver should use same .class file version. After serialization if there is any change in .class file at receiver side then receiver unable to deserialize.
* We can solve the above problem by configuring our own SerialVersionUID. We can configure our own SerialVersionUID as follows:
* **private static final long SerialVersionUID=10l;**

1. ***What is* @RequestBody*and* @ResponseBody*?***

@RequestBody**annotation maps the**HttpRequest**body to a transfer or domain object, enabling automatic deserialization** of the inbound HttpRequest body onto a Java object. Spring automatically deserializes the JSON into a Java type, assuming an appropriate one is specified.

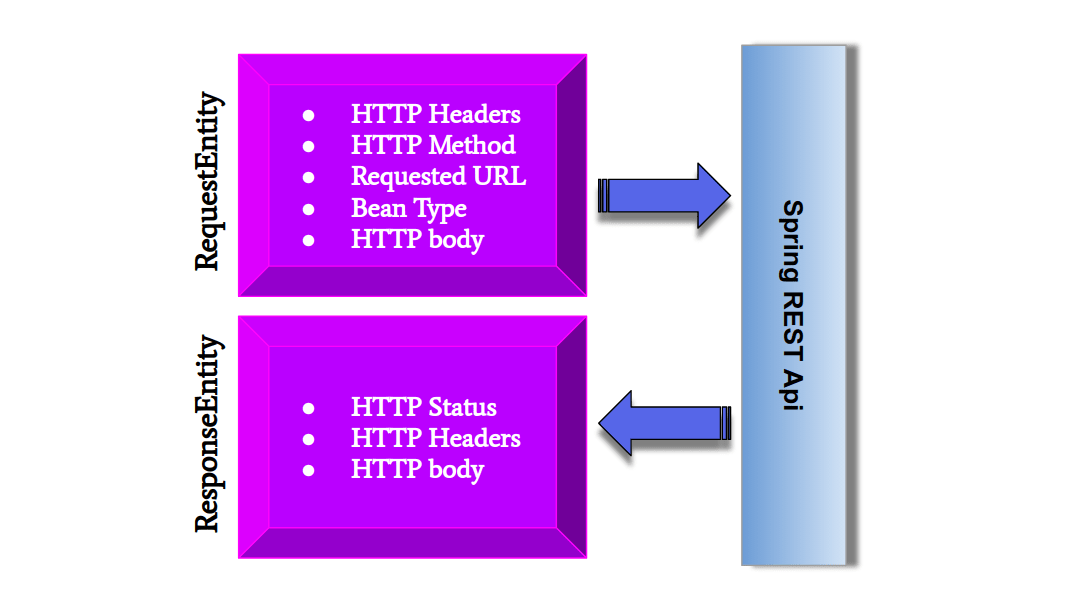
By default**, the type we annotate with the**@RequestBody**annotation must correspond to the JSON sent from our client-side controller:**

@ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

1. **How to use ResponseEntity and Requestentity in Spring?**

ResponseEntity, RequestEntity are used in Spring REST apis, RequestEntity is used as method level argument and ResponseEntity is used as method response. Both of these, can wrap any type of bean as HTTP body and provides out of the box features.

**ResponseEntity and RequestEntity both are the extensions of HttpEntity. ResponseEntity represents an HTTP response including status, headers and body, whereas RequestEntity wraps the request inside it and exposes the additional information of HTTP method and the target url.**



ResponseEntity & RequestEntity parts

# What are the differences between StackOverflowError and OutOfMemoryError in Java?

Whenever we run a java program, the operating system allocates some memory to JVM. JVM divides this memory into two parts. One is Stack memory and another is Heap memory. A stack is used for the execution of methods and heap is used to store the objects. When the Stack becomes full, JVM throws java.lang.StackOverflowError and when the heap becomes full, JVM throws java.lang.OutOfMemoryError.

## StackOverflowError

* A stack is used for the execution of methods. For every method call, one block is created in the stack memory
* The data related to the method like parameters, local variables or references to objects are stored in this block.
* When the method finishes its execution, this block is removed from the stack along with data stored in it.
* Whenever we call a method, it must finish its execution and leave the stack memory.
* If methods are staying in the stack then the stack will be full and JVM will throw java.lang.StackOverflowError.

## OutOfMemoryError

* The objects we created in Java are stored in the heap memory. When the objects are no more required, they must be removed from the memory.
* The garbage collector removes the unwanted objects from the heap memory.
* If our objects have live references, the garbage collector doesn’t remove them. It removes only those objects which don’t have live references.
* If there is no space left for new objects in the heap memory then JVM will throw java.lang.OutOfMemoryError.

1. **How linked list is replaced with binary tree in hashmap?**

Hash collisions have negative impact on the lookup time of HashMap. When multiple keys end up in the same bucket, then values along with their keys are placed in a linked list. In case of retrieval, linked list has to be traversed to get the entry. In worst case scenario, when all keys are mapped to the same bucket, the lookup time of HashMap increases from O(1) to O(n).

Java 8 has come with the following **improvements/changes** of HashMap objects in case of high collisions.

* The alternative String hash function added in Java 7 has been removed.
* Buckets containing a large number of colliding keys will store their entries in a balanced tree instead of a linked list after certain threshold is reached.

Above changes ensure performance of O(log(n)) in worst case scenarios (hash function is not distributing keys properly) and O(1) with proper **hashCode()**.

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

This JDK 8 change applies only to **HashMap, LinkedHashMap** and **ConcurrentHashMap**.

Based on a simple experiment of creating HashMaps of different sizes and performing put and get operations by key, the following results have been recorded.

1. What is OAuth?

OAuth (Open Authorization) is an open standard authorization framework for token-based authorization on the internet. OAuth, which is pronounced "oh-auth," enables an end user's account information to be used by third-party services, such as Facebook and Google, without exposing the user's account credentials to the third party. It acts as an intermediary on behalf of the end user, providing the third-party service with an access token that authorizes specific account information to be shared. The process for obtaining the token is called an authorization flow.

1. Difference between Hashmap and concurrentHashMap

|  |  |  |
| --- | --- | --- |
| Property | *java.util.****HashMap*** | *java.util.concurrent.* ***ConcurrentHashMap*** |
| synchronization | HashMap is **not synchronized.** | ConcurrentHashMap is **synchronized**. |
| 2 threads on same Map object can access it at concurrently? | Yes, because HashMap is not synchronized**.** | Yes.  But how despite of being synchronized, 2 threads on same *ConcurrentHashMap* object can access it at same time?  *ConcurrentHashMap* is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently. |
| Performance | We will **synchronize HashMap and then compare its performance with ConcurrentHashMap**.  *We can synchronize hashMap by using Collections’s class* ***synchronizedMap*** *method.*   |  | | --- | | *Map synchronizedMap = Collections.****synchronizedMap****(hashMap);* |   *Now, no 2 threads can access same instance of map concurrently.*  **Hence synchronized HashMap’s performance is slower as compared to ConcurrentHashMap.**  But why we didn’t compared HashMap (unSynchronized) with ConcurrentHashMap?  Because performance of unSynchronized collection is always better than some synchronized collection. As, default (unSynchronized) hashMap didn’t cause any locking. | **ConcurrentHashMap’s performance is faster as compared to HashMap (**because it is divided into segments, as discussed in above point**).**  *SCROLL BELOW FOR* ***PERFORMANCE COMPARISON*** *WITH DIAGRAMS.* |
| Null keys and values | HashMap allows to store **one null key** and **many null values** i.e. any key can have null value. | ConcurrentHashMap does **not allow to store null key or null value**.  Any attempt to store null key or value throws runtimeException (NullPointerException). |
| iterators | The iterators returned by the iterator() method of HashMap are [***fail-fast***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) *>*  *hashMap.keySet().iterator()*  *hashMap.values().iterator()*  *hashMap.entrySet().iterator()*  all three iterators are ***fail-fast*** | iterators are [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)*.*  *concurrentHashMap.keySet().iterator()*  *concurrentHashMap.values().iterator()*  *concurrentHashMap.entrySet().iterator()*  all three iterators are ***fail-safe.*** |
| **putIfAbsent** | HashMap does not contain putIfAbsent method.  ***putIfAbsent*** *method is equivalent to writing following code >*   |  | | --- | | **synchronized** (map){  **if** (!*map*.containsKey(key))  **return** *map*.put(key, value);  **else**  **return** *map*.get(key);  } |   [**Program to create method that provides functionality similar to putIfAbsent method of ConcurrentHashMap and to be used with HashMap**](http://www.javamadesoeasy.com/2015/04/program-to-create-method-that-provides.html) | If map does not contain specified **key**, put specified **key-value** pair in map and return null.  If map already contains specified **key**, return value corresponding to specified **key**.    [**Program to use ConcurrentHashMap’s putIfAbsent method**](http://www.javamadesoeasy.com/2015/04/program-to-use-concurrenthashmaps.html) |
| Introduced  in which java version | HashMap was introduced in **java 2 i.e. JDK 1.2**, | ConcurrentHashMap was introduced in **java 5** i.e. **JDK 1.5**, since its introduction Hashtable has become obsolete, because of concurrency level its performance is better than Hashtable. |
| Implements which interface | HashMap implements **java.util.**[**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) | ConcurrentHashMap implements  **java.util.Map** and  **java.util.concurrent.ConcurrentMap** |
| Package | HashMap is in **java.util** package | ConcurrentHashMap is in **java.util.concurrent** package. |

1. **Difference Between map() And flatMap() In Java Stream**

**map()**can be used where we have to map the elements of a particular collection to a certain function, and then we need to return the stream which contains the updated results.

**Example:** Multiplying All the elements of the list by 3 and returning the updated list.

**flatMap()** can be used where we have to flatten or transform out the string, as we cannot flatten our string using map().

**Example:** Getting the 1st Character of all the String present in a List of Strings and returning the result in form of a stream.

**Difference Between map() and flatmap()**

| map() | flatMap() |
| --- | --- |
| The function passed to map() operation returns a single value for a single input. | The function you pass to flatmap() operation returns an arbitrary number of values as the output. |
| One-to-one mapping occurs in map(). | One too many mapping occurs in flatMap(). |
| Only perform the mapping. | Perform mapping as well as flattening. |
| Produce a stream of value. | Produce a stream of stream value. |
| map() is used only for transformation. | flatMap() is used both for transformation and mapping. |

Below are the Java Programs using map() function:

* Java

|  |
| --- |
| // Java program using map() function  import java.io.\*;  import java.util.\*;  import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;  class GFG {        public static void main(String[] args)      {          // making the array list object          ArrayList<String> fruit = new ArrayList<>();          fruit.add("Apple");          fruit.add("mango");          fruit.add("pineapple");          fruit.add("kiwi");          System.out.println("List of fruit-" + fruit);            // lets use map() to convert list of fruit          List list = fruit.stream()                          .map(s -> s.length())                          .collect(Collectors.toList());          System.out.println("List generated by map-" + list);      }  } |

**Output:**

List of fruit-[Apple, mango, pineapple, kiwi]

List generated by map-[5, 5, 9, 4]

Below is the Java Program using flatMap():

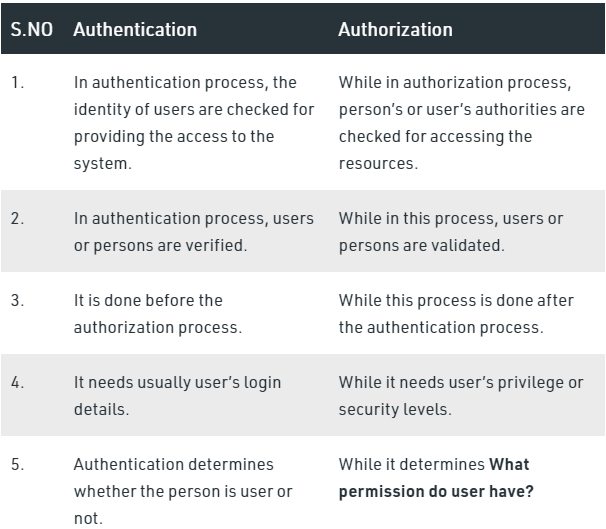
* Java

|  |
| --- |
| // Java program using flatMap() function  import java.io.\*;  import java.util.\*;  import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;  class GFG {      public static void main(String[] args)      {          // making the arraylist object of List of Integer          List<List<Integer> > number = new ArrayList<>();            // adding the elements to number arraylist          number.add(Arrays.asList(1, 2));          number.add(Arrays.asList(3, 4));          number.add(Arrays.asList(5, 6));          number.add(Arrays.asList(7, 8));            System.out.println("List of list-" + number);            // using flatmap() to flatten this list          List<Integer> flatList              = number.stream()                    .flatMap(list -> list.stream())                    .collect(Collectors.toList());            // printing the list          System.out.println("List generate by flatMap-"                             + flatList);      }  } |

**Output**

List of list-[[1, 2], [3, 4], [5, 6], [7, 8]]

List generate by flatMap-[1, 2, 3, 4, 5, 6, 7, 8]



**Java 8 Stream Intermediate Vs Terminal Operations**

|  |  |
| --- | --- |
| **Intermediate Operations** | **Terminal Operations** |
| They return stream. | They return non-stream values. |
| They can be chained together to form a pipeline of operations. | They can’t be chained together. |
| Pipeline of operations may contain any number of intermediate operations. | Pipeline of operations can have maximum one terminal operation, that too at the end. |
| Intermediate operations are lazily loaded. | Terminal operations are eagerly loaded. |
| They don’t produce end result. | They produce end result. |
| Examples : filter(), map(), distinct(), sorted(), limit(), skip() | Examples : forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny() |

**Determine if a string has all Unique Characters using stream**

**How maven works internally**

**Different maven commands**

How autowiring works in stream

*Docker is an open source platform that’s used to build, ship and run distributed services. Kubernetes is an open source orchestration platform for automating deployment, scaling and the operations of application containers across clusters of hosts.*

1. Docker Composer: You would need it where you have one or more containers that support one use case. It allows you to define all your images and containers. It defines various ports or allies where containers can talk to each other. And using docker machine you can run this on the cloud.

class TestClass {

public static String reverseString(String s){

char c1[]=s.toCharArray();

String reverse="";

for(int i=c1.length-1;i>=0;i--){

reverse=reverse+c1[i];

}

return reverse;

}

public static void main(String args[] ) throws Exception {

String a="Name";

System.out.println(reverseString(a));

}

}

Mockito is a java based mocking framework, used in conjunction with other testing frameworks such as [JUnit](https://www.journaldev.com/20834/junit5-tutorial) and [TestNG](https://www.journaldev.com/21219/testng-tutorial).

It internally uses [Java Reflection](https://www.journaldev.com/1789/java-reflection-example-tutorial) API and allows to create objects of a service. A mock object returns a dummy data and avoids external dependencies. It simplifies the development of tests by mocking external dependencies and apply the mocks into the code under test.

String s1 = new String("Test");String s2 = "Test";String s3 = s1.intern();syso(s1==s3);syso(s2==s3);

