Why String is Immutable or Final in Java?

String Constant Pool/ efficiency

If string is not immutable, changing the string with one reference will lead to the wrong value for the other references.

Security

String is widely used as parameter for many java classes, e.g. network connection, opening files, etc. Were String not immutable, a connection or file would be changed and lead to serious security threat

String is Immutable in Java because String objects are **cached in String pool**. Since cached String literals are **shared between multiple clients** there is always a risk, where one client's action would affect all another client.

`String` is immutable for several reasons, here is a summary:

- **Security**: parameters are typically represented as `String` in network connections, database connection urls, usernames/passwords etc. If it were mutable, these parameters could be easily changed.
- **Synchronization** and concurrency: making String immutable automatically makes them thread safe thereby solving the synchronization issues.
- **Caching**: when compiler optimizes your String objects, it sees that if two objects have same value (a="test", and b="test") and thus you need only one string object (for both a and b, these two will point to the same object).
- **Class loading**: `String` is used as arguments for class loading. If mutable, it could result in wrong class being loaded (because mutable objects change their state).

That being said, immutability of `String` only means you cannot change it using its public API. You can in fact bypass the normal API using reflection. See the answer **here**.
In your example, if `String` was mutable, then consider the following example:

```
String a="stack";
System.out.println(a);//prints stack
a.setValue("overflow");
System.out.println(a);//if mutable it would print overflow
```

Immutable Objects in Java

Immutability is often presented as a key concept of functional programming. Most functional programming languages like Haskell, OCaml and Scala follow a immutable-by-default approach for variables in their programs. In fact to write code which uses mutation programmers have to go out of the way and do something special - like using monads in

Haskell and mutable references in OCaml and Scala. It is of course also possible to create and use immutable objects in other programming languages. In order to get the benefits of immutability in Java we can use the following guidelines while programming -

- Mark the class **final**
- Mark all the fields **private** and **final**
- Force all the callers to construct an object of the class directly, i.e. do not use any **setter** methods
- Do not change the state of the objects in any methods of the class

Doing so will often restrict the way you can call the class and its methods. It will also make you redesign your software and rethink your algorithms. However, there are many benefits of programming with immutable objects.

1. Immutable objects are thread-safe so you will not have any synchronization issues.
2. Immutable objects are good **Map** keys and **Set** elements, since these typically do not change once created.
3. Immutability makes it easier to write, use and reason about the code (class invariant is established once and then unchanged)
4. Immutability makes it easier to parallelize your program as there are no conflicts among objects.
5. The internal state of your program will be consistent even if you have exceptions.
6. References to immutable objects can be cached as they are not going to change.

As a good programming practice in Java one should try to use immutable objects as far as possible. Immutability can have a performance cost, since when an object cannot be mutated we need to copy it if we want to write to it. When you care a lot about performance (e.g. programming a game) it may be necessary to use a mutable object. Even then it is often better to try to limit the mutability of objects. This recommendation can be summarized in the following adage by Joshua Bloch (taken from the book Effective Java) -

Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible.

If you want to further see how immutability helps in reasoning about programs you can check out HIPimm. It is a verifier for imperative C-like programs, there are examples on the site that show the use of immutability annotations.
https://www.linkedin.com/pulse/20140528113353-16837833-6-benefits-of-programming-with-immutable-objects-in-java

http://java-questions.com/garbagecollection-interview-questions.html

Q1) What is an immutable class?

Ans) Immutable class is a class which once created; its contents cannot be changed. Immutable objects are the objects whose state cannot be changed once constructed. E.g. String class

Q2) How to create an immutable class?

Ans) To create an immutable class following steps should be followed:

1. Create a final class.
2. Set the values of properties using constructor only.
3. Make the properties of the class final and private
4. Do not provide any setters for these properties.
5. If the instance fields include references to mutable objects, don't allow those objects to be changed:
    1. Don't provide methods that modify the mutable objects.
    2. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create

copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

```
public final class FinalPersonClass {
    private final String name;
    private final int age;

    public FinalPersonClass(final String name, final int age) {
     this.name = name;
     this.age = age;
    }
    public int getAge() {
      return age;
    }
    public String getName() {
     return name;
    }
  }
```

Q3) Immutable objects are automatically thread-safe –true/false?

Ans) True. Since the state of the immutable objects can not be changed once they are created they are automatically synchronized/thread-safe.

Q4) Which classes in java are immutable?

Ans) All wrapper classes in java.lang are immutable –
String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger

Q5) What are the advantages of immutability?

Ans)

- Immutable objects are automatically thread-safe, the overhead caused due to use of synchronisation is avoided.
- Once created the state of the immutable object can not be changed so there is no possibility of them getting into an inconsistent state.
- The references to the immutable objects can be easily shared or cached without having to copy or clone them as there state can not be changed ever after construction.
- The best use of the immutable objects is as the keys of a map.