# String is immutable for several reasons

## 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:

Immutability 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.

## 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**](http://stackoverflow.com/questions/20945049/is-a-java-string-really-immutable).

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

# Strings are used in [java classloader](http://www.journaldev.com/349/java-interview-questions-understanding-and-extending-java-classloader) and immutability provides security that correct class is getting loaded by Classloader. For example, think of an instance where you are trying to load java.sql.Connection class but the referenced value is changed to myhacked.Connection class that can do unwanted things to your database.

Since String is immutable, its **hashcode** is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for key in a Map and it’s processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

# Rules of serialization in Java

* An object is serializable only if its class or its superclass implements the Serializable(or  Externalizable) interface.
* An object is serializable (itself implements the Serializable interface) even if its superclass is not. However, the first superclass in the hierarchy of the serializable class, that does not implements Serializable interface, MUST have a no-arg constructor. If this is violated, readObject() will produce a java.io.InvalidClassException in runtime.
* The no-arg contructor of every non-serializable superclass will run when an object is deserialized. However, the deserialized objects? constructor does not run when it is deserialized.
* The class must be visible at the point of serialization.
* All primitive types are serializable.
* Transient fields (with transient modifier) are NOT serialized, (i.e., not saved or restored). A class that implements Serializable must mark transient fields of classes that do not support serialization (e.g., a file stream).
* Static fields (with static modifier) are Not serialized.
* If member vairiables of a serializable object reference to a non-serializable object, the code will compile but a RumtimeException will be thrown.

# Memocry usage of Java Strings and string-related objects