**Why String is immutable or final in Java**

1) Imagine StringPool facility without making string immutable , its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many [reference variables](http://javarevisited.blogspot.sg/2012/02/difference-between-instance-class-and.html) , so if any one of them change the value others will be automatically gets affected i.e. lets say  
  
String A = "Test"  
String B = "Test"   
  
Now String B called "Test".toUpperCase() which change the same object into "TEST" , so A will also be "TEST" which is not desirable.  
  
2)String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string , you can pass database URL as string for opening database connection, you can [open any file in Java](http://javarevisited.blogspot.sg/2012/07/read-file-line-by-line-java-example-scanner.html) by passing name of file as argument to File I/O classes.  
  
In case, if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access of those file. Because of immutability, you don't need to worry about those kind of threats. This reason also gel with, **Why String is final in Java**, by making java.lang.String final, Java designer ensured that no one overrides any behavior of String class.  
  
3)Since String is immutable it can safely shared between many threads ,which is very important for multithreaded programming and to avoid any [synchronization issues in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), Immutability also makes String instance [thread-safe in Java](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), means you don't need to synchronize String operation externally. Another important point to note about String is [memory leak caused by SubString](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html), which is not a thread related issues but something to be aware of.  
  
4) Another reason of **Why String is immutable in Java** is to **allow String to cache its hashcode** , being immutable String in Java caches its hashcode, and do not calculate every time we call hashcode method of String, which makes it very fast as hashmap key to be used in [hashmap in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html).  This one is also suggested by  Jaroslav Sedlacek in comments below. In short because String is immutable, no one can change its contents once created which guarantees [hashCode](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html)of String to be same on multiple invocation.  
  
5) Another good reason of Why String is immutable in Java suggested by Dan Bergh Johnsson on comments is: The absolutely most important reason that String is immutable is that it is used by the [class loading mechanism](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html), and thus have profound and fundamental security aspects. Had String been mutable, a request to load "java.io.Writer" could have been changed to load "mil.vogoon.DiskErasingWriter"  
  
  
Security and String pool being primary reason of making String immutable, I believe there could be some more very convincing reasons as well, Please post those reasons as comments and I will include those on this post. By the way, above reason holds good to answer, another [Java interview questions](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) **"Why String is final in Java".**

**How SubString method works in Java - Substring Memory Leak**

substring method is used to get parts of String in Java. It’s defined in java.lang.String class, and it's an [overloaded method](http://javarevisited.blogspot.com/2011/12/method-overloading-vs-method-overriding.html). One version of substring method takes just beginIndex, and returns part of String started from beginIndex till end, while other takes two parameters, beginIndex and endIndex, and returns part  of String starting frombeginIndex to endIndex-1. He also stressed that every time you call  substring() method in Java,  it will return a new String because [String is immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html).

**what will happen if beginIndex is equal to length in substring(int beginIndex)**, no it won't throw IndexOutOfBoundException instead it will return [empty String](http://javarevisited.blogspot.com/2013/02/5-ways-to-check-if-string-is-empty-in-java-examples.html). Same is the case when beginIndex and endIndex is equal, in case of second method. It will only throw StringIndexBoundException when beginIndex is negative, larger thanendIndex or larger than length of String.  
Most of Java developers fail here, because they don't know how exactly substring method works, until they have not seen the code of java.lang.String. If you look substring method inside String class, you will figure out that it calls String (int offset, int count, char value []) [constructor](http://javarevisited.blogspot.com/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) to create new String object. What is interesting here is, value[], which is the same character array used to represent original string. So **what's wrong with this**?

In case If you have still not figured it out, If the original string is very long, and has array of size 1GB, no matter how small a substring is, it will hold 1GB array.  This will also stop original string to be [garbage collected](http://javarevisited.blogspot.com/2012/10/10-garbage-collection-interview-question-answer.html), in case if doesn't have any live reference. This is clear case of memory leak in Java, where memory is retained even if it's not required. That's how *substring method* creates **memory leak**.

## How SubString in Java works

Obviously next question from interviewer would be,  **how do you deal with this problem?** Though you can not go, and change Java substring method, you can still make some work around, in case you are creating substring of significant longer String. Simple solution is to trim the string, and keep size of character array according to length of substring. Luckily java.lang.String has constructor to do this, as shown in below example.

*// comma separated stock symbols from NYSE*

String listOfStockSymbolsOnNYSE = getStockSymbolsForNYSE();

*//calling String(string) constructor*

String apple = new String(listOfStockSymbolsOnNYSE.substring(appleStartIndex,appleEndIndex));

If you look code on java.lang.String class, you will see that this [constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) trim the array, if it’s bigger than String itself.

public String(String original) {

...

if (originalValue.length &gt; size) {

*// The array representing the String is bigger than the new*

*// String itself. Perhaps this constructor is being called*

*// in order to trim the baggage, so make a copy of the array.*

int off = original.offset;

v = Arrays.copyOfRange(originalValue, off, off+size);

} else {

*// The array representing the String is the same*

*// size as the String, so no point in making a copy.*

v = originalValue;

}

...

}

Another way to solve this problem is to call intern() method on substring, which will than fetch an [existing string](http://javarevisited.blogspot.sg/2012/03/how-to-compare-two-string-in-java.html) from pool or add it if necessary. Since the String in the pool is a real string it only take space as much it requires. It’s also worth noting that sub-strings are not internalized, when you call intern() method on original String. Most developer successfully answers first three questions, which is related to usage of substring, but they get stuck on last two, How substring creates memory leak or How substring works. It's not completely there fault, because what you know is that every time substring() returns new String which is not exactly true, since it’s backed by same [character array](http://javarevisited.blogspot.com/2012/02/how-to-convert-char-to-string-in-java.html).

This was the only interview question, which bothers my friend little otherwise, its standard service level company  Java interview in India. By the way, he got the call a day after ,even though he struggled little bit on How SubString method works in Java, and that was the reason he shared this interview experience with me.

**Why character array is better than String for Storing password in Java**1) Since **Strings are immutable in Java** if you store password as plain text it will be available in memory until Garbage collector clears it and since String are used in String pool for reusability there is pretty high chance that it will be remain in memory for long duration, which pose a security threat. Since any one who has access to memory dump can find the password in clear text and that's another reason you should always used an encrypted password than plain text. Since Strings are immutable there is no way contents of Strings can be changed because [any change will produce new String](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html), while if you char[] you can still set all his element as blank or zero. So **Storing password in character array clearly mitigates security risk of stealing password**.

2) **Java itself recommends**using getPassword() method of JPasswordField which returns a char[] and deprecated getText() method which returns password in clear text stating security reason. Its good to follow advice from Java team and adhering to standard rather than going against it.

3) With String there is always a risk of printing plain text in [log file or console](http://javarevisited.blogspot.com/2011/05/top-10-tips-on-logging-in-java.html) but if use [Array](http://javarevisited.blogspot.com/2012/01/anonymous-array-example-java-create.html) you won't print contents of array instead its memory location get printed. though not a real reason but still make sense.

[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) strPassword="Unknown";  
**char**[] charPassword= **new** **char**[]{'U','n','k','w','o','n'};  
**System**.out.println("String password: " + strPassword);  
**System**.out.println("Character password: " + charPassword);  
  
**String** password: Unknown  
**Character** password: [C@110b053

That's all on *why character array is better choice than String for storing passwords in Java*.  Though using char[] is not just enough you need to erase content to be more secure. I also suggest working with hash'd or [encrypted password](http://javarevisited.blogspot.com/2012/02/how-to-encode-decode-string-in-java.html) instead of plaintext and clearing it from memory as soon as authentication is completed.