Why is String immutable in Java?

I couldn't understand the reason of it. I always use String class like other developers, but when I modify the value of it, I need to create new instance of String.

What might be the reason of immutability for String class in Java?

I know there are some alternatives like StringBuffer or StringBuilder. It's just curiosity.

java | programming-languages | language-design

edited Apr 18 '13 at 14:27

asked Apr 16 '13 at 12:27

blank
617 1 7 22

19 Technically, it's not a duplicate, but Eric Lippert gives a great answer to this question here: programmers.stackexchange.com/a/190913/33843 – Heinzi Apr 16 '13 at 13:40

1 @Heinzi I agree. – blank Apr 16 '13 at 13:48

9 Answers

Concurrency

Java was defined from the start with considerations of concurrency. As often been mentioned shared mutables are problematic. One thing can change another behind the back of another thread without that thread being aware of it.

There are a host of multithreaded C++ bugs that have croped up because of a shared string - where one module thought it was safe to change when another module in the code had saved a pointer to it and expected it to stay the same.

The 'solution' to this is that every class makes a defensive copy of the mutable objects that are passed to it. For mutable strings, this is O(n) to make the copy. For immutable strings, making a copy is O(1) because it isn't a copy, its the same object that can't change.

In a multithreaded environment, immutable objects can always be safely shared between each other. This leads to an overall reduction in memory usage and improves memory caching.

Security

Many times strings are passed around as arguments to constructors - network connections and protocals are the two that most easily come to mind. Being able to change this at an undetermined time later in the execution can lead to security issues (the function thought it was connecting to one machine, but was diverted to another, but everything in the object looks like it connected to the first... its even the same string).

Java lets one use reflection - and the parameters for this are strings. The danger of one passing a string that can get modified through the way to another method that reflects. This is very bad.

Keys to the Hash

The hash table is one of the most used data structures. The keys to the data structure are very often strings. Having immutable strings means that (as above) the hash table does not need to make a copy of the hash key each time. If strings were mutable, and the hash table didn't make this, it would be possible for something to change the hash key at a distance.

The way the Object in java works, is that everything has a hash key (accessed via the hashCode() method). Having an immutable string means that the hashCode can be cached. Considering how often Strings are used as keys to a hash, this provides a significant performance boost (rather than having to recalculate the hash code each time).

Substrings

By having the String be immutable, the underlying character array that backs the data structure is also immutable. This allows for certain optimizations on the `substring` method the be done (they aren't *necessarily* done - it also introduces the possibility of some memory leaks too).

If you do:

```
String foo = "smiles";
String bar = foo.substring(1,5);
```

The value of `bar` is 'mile'. However, both `foo` and `bar` can be backed by the same character array, reducing the instantiation of more character arrays or copying it - just using different start and end points within the string.

```
foo |    | (0, 6)
    v    v
    smiles
     ^  ^
bar  |  |  (1, 5)
```

Now, the downside of that (the memory leak) is that if one had a 1k long string and took the substring of the first and second character, it would also be backed by the 1k long character array. This array would remain in memory even if the original string that had a value of the entire character array was garbage collected.

One can see this in http://www.docjar.com/html/api/java/lang/String.java.html

```
    public String(char value[], int offset, int count) {
        if (offset < 0) {
            throw new StringIndexOutOfBoundsException(offset);
        }
        if (count < 0) {
            throw new StringIndexOutOfBoundsException(count);
        }
        // Note: offset or count might be near -1>>>1.
        if (offset > value.length - count) {
            throw new StringIndexOutOfBoundsException(offset + count);
        }
        this.offset = 0;
        this.count = count;
        this.value = Arrays.copyOfRange(value, offset, offset+count);
    }

    // Package private constructor which shares value array for speed.
    String(int offset, int count, char value[]) {
        this.value = value;
        this.offset = offset;
        this.count = count;
    }

    public String substring(int beginIndex, int endIndex) {
        if (beginIndex < 0) {
            throw new StringIndexOutOfBoundsException(beginIndex);
        }
        if (endIndex > count) {
            throw new StringIndexOutOfBoundsException(endIndex);
        }
        if (beginIndex > endIndex) {
            throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
        }
        return ((beginIndex == 0) && (endIndex == count)) ? this :
            new String(offset + beginIndex, endIndex - beginIndex, value);
    }
```

Note how the substring uses the package level String constructor that doesn't involve any copying of the array and would be much faster (at the expense of possibly keeping around some large arrays - though not duplicating large arrays either).

Do note that the above code is for Java 1.6. The way the substring constructor is implemented was changed with Java 1.7 as documented in Changes to String internal representation made in Java 1.7.0_06 - the issue bing that memory leak that I mentioned above. Java likely wasn't seen as being a language with lots of String manipulation and so the performance boost for a substring was a good thing. Now, with huge XML documents stored in strings that are never collected, this becomes an issue... and thus the change to the `String` not using the same underlying array with a substring, so that the larger character array may be collected more quickly.

Don't abuse the Stack

One *could* pass the value of the string around instead of the reference to the immutable string to avoid issues with mutability. However, with large strings, passing this on the stack would be...

abusive to the system (putting entire xml documents as strings on the stack and then taking them off or continuing to pass them along...).

The possibility of deduplication

Granted, this wasn't an initial motivation for why Strings should be immutable, but when one is looking at the rational of why immutable Strings are a good thing, this is certainly something to consider.

Anyone who has worked with Strings a bit knows that they can suck memory. This is especially true when you're doing things like pulling data from databases that sticks around for awhile. Many times with these stings, they are the same string over and over again (once for each row).

> Many large-scale Java applications are currently bottlenecked on memory. Measurements have shown that roughly 25% of the Java heap live data set in these types of applications is consumed by String objects. Further, roughly half of those String objects are duplicates, where duplicates means string1.equals(string2) is true. Having duplicate String objects on the heap is, essentially, just a waste of memory. ...

With Java 8 update 20, JEP 192 (motivation quoted above) is being implemented to address this. Without getting into the details of how string deduplication works, it is essential that the Strings themselves are immutable. You can't deduplicate StringBuilders because they can change and you don't want someone changing something from under you. Immutable Strings (related to that String pool) means that you can go through and if you find two strings that are the same, you can point one string reference to the other and let the garbage collector consume the newly unused one.

Other languages

Objective C (which predates Java) has `NSString` and `NSMutableString` .

C# and .NET made the same design choices of the default string being an immutable.

Lua strings are also immutable.

Historically, Lisp, Scheme, Smalltalk all intern the string and thus have it be immutable. More modern dynamic languages often use strings in some way that requires that they be immutable (it may not be a *String*, but it is immutable).

Conclusion

These design considerations have been made again and again in a multitude of languages. It is the general consensus that immutable strings, for all of their awkwardness, are better than the alternatives and lead to better code (fewer bugs) and faster executables overall.

edited Sep 20 at 2:29

answered Apr 16 '13 at 18:39

MichaelT

34.8k 13 85 127

Thanks for the detailed explanation. – blank Apr 16 '13 at 20:25

3 Java provides mutable and immutable strings. This answer details some performance advantages that can be done on immutable strings, and some reasons one might choose immutable data; but does not discuss why the immutable version is the default version. – Billy ONeal Apr 17 '13 at 6:23

2 @BillyONeal: a secure default and an unsecure alternative *almost always* leads to more secure systems than the opposite approach. – Joachim Sauer Apr 17 '13 at 14:31

2 @BillyONeal If the immutable wasn't the default the issues of concurrency, security, and hashes would be more commonplace. The language designers have chosen (in part in response to C) to make a language where the defaults are set up to try to prevent a number of common bugs to try to improve programmer efficiency (not having to worry about these bugs anymore). There are fewer bugs (obvious and hidden) with immutable strings than with mutable ones. – MichaelT Apr 17 '13 at 14:32

@Joachim: I'm not claiming otherwise. – Billy ONeal Apr 17 '13 at 17:23

Reasons I can recall :

1. String Pool facility without making string immutable is not possible at all because in case of string pool one string object/literal e.g. "XYZ" will be referenced by many reference variables , so if any one of them changes the value others will be automatically gets affected .
2. String has been widely used as parameter for many java classes e.g. for opening network connection , for opening database connection, opening files . If String is not immutable, this would lead to serious security threat .
3. Immutability allows String to cache its hashcode.

4. Makes it thread-safe .

answered Apr 16 '13 at 12:56

NINCOMPOOP

389 1 14

The Java Virtual Machine performs several optimizations regarding string operations which couldn't be performed otherwise. For example, if you had a string with value "Mississippi" and you assigned "Mississippi".substring(0, 4) to another string, as far as you know, a copy was made of the first four characters to make "Miss". What you don't know is that both share the same original string "Mississippi" with one being the owner and the other being a reference of that string from position 0 to 4. (The reference to the owner prevents the owner from being collected by the garbage collector when the owner goes out of scope)

This is trivial for a string as small as "Mississippi", but with larger strings and multiple operations, not having to copy the string is a big time saver! If strings were mutable, then you could not do this, because modifying the original would affect the substring "copies" as well.

Also, as Donal mentions, the advantage would be greatly weighed down by its disadvantage. Imagine that you write a program which depends on a library and you use a function which returns a string. How could you be sure that that value will remain constant? To ensure no such thing happens, you'd always have to produce a copy.

What if you have two threads sharing the same string? You wouldn't want to be reading a string that is currently being rewritten by another thread, would you? String would then therefore have to be thread safe, which being the common class that it is, would make virtually every Java program that much slower. Otherwise, you'd have to make a copy for every thread which requires that string or you would have to put the code using that string in a synchronize block, both of which only slow down your program.

For all these reasons, that was one of the early decisions made for Java in order to differentiate itself from C++.

answered Apr 16 '13 at 12:48

Neil

8,609 16 35

Theoretically, you can do a multi-layer buffer management that allows for copy-on-mutation-if-shared, but that's very hard to make work efficiently in a multi-threaded environment. – Donal Fellows Apr 16 '13 at 13:29

@DonalFellows I just assumed that since the Java Virtual Machine isn't written in Java (obviously), it is managed internally using shared pointers or something of the like. – Neil Apr 16 '13 at 13:32

Immutability means that constants held by classes that you don't own can't be modified. Classes that you don't own include those that are in the core of the implementation of Java, and strings that shouldn't be modified include things like security tokens, service addresses, etc. You *really* shouldn't be able to modify those sorts of things (and this applies doubly when operating in sandboxed mode).

If String wasn't immutable, every time you retrieved it from some context which didn't want the string's contents changed under its feet, you'd have to take a copy “just in case”. That gets very expensive.

answered Apr 16 '13 at 12:29

Donal Fellows

4,738 11 29

3 This exact same argument applies to *any* type, not just to `String` . But, for example, `Array` s are mutable nonetheless. So, why are `String` s immutable and `Array` s not. And if immutability is so important, then why does Java make it so hard to create and work with immutable objects? – Jörg W Mittag Apr 16 '13 at 12:45

@JörgWMittag: I assume that's basically a question of how radical they wanted to be. Having an immutable String was pretty radical, back in the Java 1.0 days. Having a (primarily or even exclusively) immutable collection framework as well, could have been too radical for getting wide usage of the language. – Joachim Sauer Apr 16 '13 at 13:04

Doing an effective immutable collections framework is quite tricky to make performant, speaking as someone who has written such a thing (but not in Java). I also wish totally that I had immutably arrays; that would have saved me quite a bit of work. – Donal Fellows Apr 16 '13 at 13:21

@DonalFellows: pcollections aims to do just that (never used it myself, however). – Joachim Sauer Apr 16 '13 at 13:23

3 @JörgWMittag: There are people (commonly from a purely functional perspective) who would argue that *all* types should be immutable. Likewise, I think that if you add up all of the issues that one deals with working with mutable state in parallel and concurrent software, you might agree that working with immutable objects is often *much* easier than mutable ones. – Steve Evers Apr 16 '13 at 14:33

The reason for string's immutability comes from consistency with other primitive types in the language. If you have an `int` containing the value 42, and you add the value 1 to it, you don't change the 42. You get a new value, 43, which is completely unrelated to the starting values. Mutating primitives other than string makes no conceptual sense; and as such programs that treat strings as immutable are often easier to reason about and understand.

Moreover, Java really provides both mutable and immutable strings, as you see with `StringBuilder`; really, only the *default* is the immutable string. If you want to pass references to `StringBuilder` around everywhere you are perfectly welcome to do so. Java uses separate types (`String` and `StringBuilder`) for these concepts because it doesn't have support for expressing mutability or lack thereof in its type system. In languages which have support for immutability in their type systems (e.g. C++'s `const`), there is often a single string type which serves both purposes.

Yes, having string be immutable allows one to implement some optimizations specific to immutable strings, such as interning, and allows passing string references around without synchronization across threads. However, this confuses the mechanism with the intended goal of a language with a simple and consistent type system. I liken this to how everyone thinks of garbage collection the wrong way; garbage collection is not "reclamation of unused memory"; it is "simulating a computer with unlimited memory". The discussed performance optimizations are things that are done to make the goal of immutable strings perform well on real machines; not the reason for such strings to be immutable in the first place.

edited Apr 19 '13 at 19:31 answered Apr 17 '13 at 6:18

Billy ONeal
5,613 3 25 52

@Billy-Oneal.. Regarding "If you have an int containing the value 42, and you add the value 1 to it, you don't change the 42. You get a new value, 43, which is completely unrelated to the starting values." Are you sure about that? – Shamit Verma Apr 28 at 11:11

@Shamit: Yes, I'm sure. Adding 1 to 42 results in 43. It does not make the number 42 mean the same thing as the number 43. – Billy ONeal Apr 28 at 17:14

@Shamit: Similarly, you can't do something like `43 = 6` and expect the number 43 to mean the same thing as the number 6. – Billy ONeal Apr 28 at 17:15

int i = 42; i = i+1; this code will store 42 in memory and then change values in the same location to 43. So in fact, variable "i" acquires a new value of 43. – Shamit Verma Apr 29 at 20:00

@Shamit: In that case, you mutated `i`, not 42. Consider `string s = "Hello "; s += "World";`. You mutated the value of variable `s`. But the strings `"Hello "`, `"World"`, and `"Hello World"` are immutable. – Billy ONeal Apr 29 at 23:16

---

1) String Pool

Java designer knows that String is going to be most used data type in all kind of Java applications and that's why they wanted to optimize from start. One of key step on that direction was idea of storing String literals in String pool. Goal was to reduce temporary String object by sharing them and in order to share, they must have to be from Immutable class. You can not share a mutable object with two parties which are unknown to each other. Let's take an hypothetical example, where two reference variable is pointing to same String object:

```
String s1 = "Java";
String s2 = "Java";
```

Now if s1 changes the object from "Java" to "C++", reference variable also got value s2="C++", which it doesn't even know about it. By making String immutable, this sharing of String literal was possible. In short, key idea of String pool can not be implemented without making String final or Immutable in Java.

2) Security

Java has clear goal in terms of providing a secure environment at every level of service and String is critical in those whole security stuff. String has been widely used as parameter for many Java classes, e.g. for opening network connection, you can pass host and port as String, for reading files in Java you can pass path of files and directory as String and for opening database connection, you

can pass database URL as String. If String was not immutable, a user might have granted to access a particular file in system, but after authentication he can change the PATH to something else, this could cause serious security issues. Similarly, while connecting to database or any other machine in network, mutating String value can pose security threats. Mutable strings could also cause security problem in Reflection as well, as the parameters are strings.

3) Use of String in Class Loading Mechanism

Another reason for making String final or Immutable was driven by the fact that it was heavily used in class loading mechanism. As String been not Immutable, an attacker can take advantage of this fact and a request to load standard Java classes e.g. java.io.Reader can be changed to malicious class com.unknown.DataStolenReader. By keeping String final and immutable, we can at least be sure that JVM is loading correct classes.

4) Multithreading Benefits

Since Concurrency and Multi-threading was Java's key offering, it made lot of sense to think about thread-safety of String objects. Since it was expected that String will be used widely, making it Immutable means no external synchronization, means much cleaner code involving sharing of String between multiple threads. This single feature, makes already complicate, confusing and error prone concurrency coding much easier. Because String is immutable and we just share it between threads, it result in more readable code.

5) Optimization and Performance

Now when you make a class Immutable, you know in advance that, this class is not going to change once created. This guarantee open path for many performance optimization e.g. caching. String itself know that, I am not going to change, so String cache its hashcode. It even calculate hashcode lazily and once created, just cache it. In simple world, when you first call hashCode() method of any String object, it calculate hash code and all subsequent call to hashCode() returns already calculated, cached value. This results in good performance gain, given String is heavily used in hash based Maps e.g. Hashtable and HashMap. Caching of hashcode was not possible without making it immutable and final, as it depends upon content of String itself.

edited Sep 20 at 2:27

MichaelT
34.8k 13 85 127

answered Feb 24 at 11:38

saidesh kilaru
31 4

Imagine a system where you accept some data, verify its correctness and then pass it on (to be stored in a DB, for example).

Assuming that data is a `String` and it has to be at least 5 characters long. Your method looks something like this:

```
public void handle(String input) {
  if (input.length() < 5) {
    throw new IllegalArgumentException();
  }
  storeInDatabase(input);
}
```

Now we can agree, that when `storeInDatabase` is called here, the `input` will fit the requirement. **But** if `String` were mutable, then the *caller* could alter the `input` object (from another thread) *right after it has been verified and before it's been stored in the database*. This would require good timing and would *probably* not go well every time, but occasionally, he'd be able to get you to store invalid values in the database.

Immutable data types are a very simply solution to this (and a lot of related) problems: whenever you check some value, you can **depend** on the fact that the checked condition is still true later on.

answered Apr 16 '13 at 13:01

Joachim Sauer
8,404 2 37 41

> Thanks for the explanation. What if i call handle method like this; handle(new String(input + "naberlan")). I guess i can store invalid values in the db like this. – blank Apr 16 '13 at 13:18

> @blank: well, since the `input` of the `handle` method is already too long (no matter what the *original* `input` is), it would simply throw an exception. You're creating a new input *before* calling the method. That's not a problem. – Joachim Sauer Apr 16 '13 at 13:21

---

1. they are expensive and keeping them immutable allows for such things as sub strings sharing the byte array of the main string. (speed boost also as do not need to make a new byte array

and copy over)

2. security - would not want your package or class code to be re named

   [removed old 3 looked at StringBuilder src - it does not share memory with string (until modified) I think that was in 1.3 or 1.4]

3. cache hashcode

4. for mutalble strings use SB (builder or buffer as needed)

edited Dec 3 '13 at 16:02

answered Apr 16 '13 at 13:19
tgkprog
461 3 14

2 1. Of course, there is the penalty of not being able to destroy the larger parts of the string if this happens. Interning is not free; though it does improve performance for many real world programs. 2. There could easily be "string" and "ImmutableString" which could satisfy that requirement. 3. I'm not sure I understand that... – Billy ONeal Apr 17 '13 at 6:20

.3. should have been cache the hash code. This too could be done with a mutable string. @billy-oneal – tgkprog Dec 3 '13 at 16:04

Strings should have been a primitive data type in Java. If they had been, then strings would default to being mutable and the final keyword would generate immutable strings. Mutable string are useful and so there are multiple hacks for mutable strings in the stringbuffer, stringbuilder, and charsequence classes.

answered Apr 16 '13 at 17:49
CWallach
392 3 2

3 This doesn't answer the "why" aspect of what it is now that the question does ask. Additionally, java final doesn't work that way. Mutable strings are not hacks, but rather real design considerations based on the most common uses of strings and the optimizations that can be done to improve the jvm. – MichaelT Apr 16 '13 at 17:56

1 The answer to the "why" is a poor language design decision. Three slightly different ways to support mutable strings is a hack that the compiler/JVM should handle. – CWallach Apr 16 '13 at 18:06

3 String and StringBuffer were the original. StringBuilder was added later recognizing a design difficulty with StringBuffer. Mutable and immutable strings being different objects is found in many languages as the design consideration was made again and again and decided that each are different objects each time. C# "Strings are immutable" and Why .NET String is immutable?, objective C NSString is immutable while NSMutableString is mutable. stackoverflow.com/questions/9544182 – MichaelT Apr 16 '13 at 18:12

---

protected by gnat Jun 29 at 18:07

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?