Java: String is Immutable. What exactly is the meaning?

Difficulty Level : Easy Last Updated : 11 Dec, 2018

Before proceeding further with the fuss of *immutability*, let's just take a look into the `String` class and its functionality a little before coming to any conclusion.

This is how `String` works:

```
String str = "knowledge";
```

This, as usual, creates a string containing `"knowledge"` and assigns it a reference `str`. Simple enough? Lets perform some more functions:

```
// assigns a new reference to the
// same string "knowledge"
String s = str;
```

Let's see how the below statement works:

```
  str = str.concat(" base");
```

This appends a string `" base"` to `str`. But wait, how is this possible, since `String` objects are immutable? Well to your surprise, it is.

When the above statement is executed, the VM takes the value of `String str`, i.e. `"knowledge"` and appends `" base"`, giving us the value `"knowledge base"`. Now, since `String`s are immutable, the VM can't assign this value to `str`, so it creates a new `String` object, gives it a value `"knowledge base"`, and gives it a reference `str`.

An important point to note here is that, while the `String` object is immutable, **its reference variable is not.** So that's why, in the above example, the reference was made to refer to a newly formed `String` object.

At this point in the example above, we have two `String` objects: the first one we created with value `"knowledge"`, pointed to by `s`, and the second one `"knowledge base"`, pointed to by `str`. But, technically, we have three `String` objects, the third one being the literal `"base"` in the `concat` statement.

Important Facts about String and Memory usage

What if we didn't have another reference `s` to `"knowledge"`? We would have lost that `String`. However, it still would have existed, but would be considered lost due to having no references.
Look at one more example below

```
/*package whatever // do not write package name here */

import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        String s1 = "java";
        s1.concat(" rules");

        // Yes, s1 still refers to "java"
        System.out.println("s1 refers to " + s1);
    }
}
```

Output:

```
s1 refers to java
```

What's happening:

1. The first line is pretty straightforward: create a new `String` `"java"` and refer `s1` to it.
2. Next, the VM creates another new `String` `"java rules"`, but nothing refers to it. So, the second `String` is instantly lost. We can't reach it.

The reference variable `s1` still refers to the original `String` `"java"`.

Almost every method, applied to a `String` object in order to modify it, creates new `String` object. So, where do these `String` objects go? Well, *these exist in memory, and one of the key goals of any programming language is to make efficient use of memory.*

As applications grow, *it's very common for* `String` *literals to occupy large area of memory, which can even cause redundancy.* So, in order to make Java more efficient, **the JVM sets aside a special area of memory called the "**String constant pool**".**

When the compiler sees a `String` literal, it looks for the `String` in the pool. If a match is found, the reference to the new literal is directed to the existing `String` and no new `String` object is created. The existing `String` simply has one more reference. Here comes the point of making `String` objects immutable:

In the `String` constant pool, a `String` object is likely to have one or many references. *If several references point to same* `String` *without even knowing it, it would be bad if one of the references modified that* `String` *value. That's why* `String` *objects are immutable.*

Well, now you could say, *what if someone overrides the functionality of* `String` *class?* That's the reason that **the `String` class is marked `final`** so that nobody can override the behavior of its methods.