

immutable

Immutable means not capable or susceptible to change, invariable, unalterable.

If an object is immutable, its contents cannot be changed once it has been created. The contents of a *mutable* object can be changed.

All objects of class `String` and the wrapper classes like `Integer` and `Character` and `Boolean` are immutable.

But consider this. Suppose `String` variable `s` points to a `String` object containing the characters “xyz”, as shown to the right. Object `s` (i.e. the object pointed to by `s`) is immutable. The string of characters in `s` cannot be changed. But when an assignment statement

```
s = s + “bc”;
```

is executed, evaluation of `s + “bc”` creates a new `String` object to hold the characters “xyzbc”, and the value of the expression—the pointer to the new object—is stored in variable `s`. Thus, the situation after execution is as shown to the right below. Note that the old object is still there, but `s` points at the new object instead of the old.

Be careful with `String` functions. The tendency is to think that the code¹

```
s.replace(‘x’, ‘$’);
```

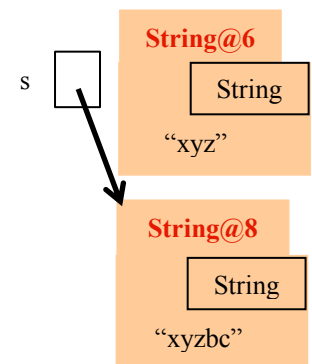
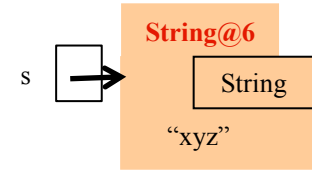
will change `s` to contain the string “\$yzbc”. But `replace` is a function, not a procedure, and this code creates a new object that contains “\$yzbc” and then throws it away—the pointer to the new object hasn’t been saved. Use the following assignment to replace ‘x’ in `s` by ‘\$’:

```
s = s.replace(‘x’, ‘$’);
```

What about execution time?

You can see that a catenation like `s + “b”` can be costly: it takes time proportional to the length of `s`. Suppose the length of `s` is 1000. All 1000 characters have to be copied into the new object. In most of the programs you write, you don’t have to worry about this issue. But a situation may arise in which the use of such a catenation within a loop drastically slows down a program.

Java does have a class `java.lang.StringBuilder` whose objects are mutable—they can be changed. A wise program faced with the slow `String` catenation will use `StringBuilder`. You can study the spec of `StringBuilder` yourself.



¹ `s.replace(c1, c2)` returns a new `String` that is like `s` except that all occurrences of character `c1` have been replaced by character `c2`. Look up the spec of `replace` and other functions in the Java API documentation.