1Z0-808 Exam Topic Reviewer

TopicId: 1013
Topic: String Immutability and Operations

August 5, 2025

The Golden Rule: Strings are Immutable

If there is one thing you must burn into your memory for the 1Z0-808 exam, it's this: **Objects of the `String` class are immutable**. This means that once a `String` object is created, its internal state (the sequence of characters) cannot be changed. Ever. Many students get tripped up by this. They see code like `myString.toUpperCase();` and assume it changes `myString`. It does not. Any method that appears to modify a `String` will always **return a new `String` object** containing the modification. The original string is left untouched.

A Classic Exam Trap

Analyze this code snippet. What does it print?

```
String name = "Java";
name.concat(" SE 8"); // This returns a new String, which is ignored.
name.toUpperCase();   // This also returns a new String, which is ignored.
System.out.println(name);
```

The output is simply **Java**. The original `name` object was never reassigned. To make the changes stick, you must reassign the reference:

```
String name = "Java";
name = name.concat(" SE 8"); // name now refers to "Java SE 8"
name = name.toUpperCase();   // name now refers to "JAVA SE 8"
System.out.println(name); // Prints: JAVA SE 8
```

1 The String Constant Pool

To save memory, the JVM maintains a special area called the String Constant Pool.

- **String Literals:** When you create a string with a literal like `String s1 = "hello";`, the JVM looks for the string "hello" in the pool. If found, it returns a reference to the existing object. If not, it creates a new `String` object in the pool and returns a reference to it.
- `new` **Keyword:** When you use `String s2 = new String("hello");`, you are explicitly telling the JVM: "*Create a brand new object in the heap memory, regardless of what's in the pool.*"

This distinction is critical for understanding the difference between `==` and `.equals()`.

`==` vs. `.equals()`

```
String s1 = "Test"; // Goes into the pool
String s2 = "Test"; // Reuses the object from the pool
String s3 = new String("Test"); // Creates a new object in the heap

System.out.println(s1 == s2);      // true: s1 and s2 refer to the same object in
System.out.println(s1 == s3);      // false: s1 is in the pool, s3 is in the heap
System.out.println(s1.equals(s3)); // true: their character sequences are identic
```

- `==`: Compares object references. It checks if two variables point to the exact same object in memory.
- `.equals()`: Compares the actual character sequences. For Strings, this is almost always what you want.

2 Essential String Methods

You must be familiar with the common `String` API methods. Remember, they all return a new string!

- `int length()`: Returns the number of characters.
- `char charAt(int index)`: Returns the character at a given index (0-based). Can throw `StringIndexOutOfBoundsException`.
- `String substring(int beginIndex, int endIndex)`: Extracts a portion of the string. The character at `beginIndex` is included, but the character at `endIndex` is **excluded**. Forgetting this is a common mistake.
- `String toLowerCase()` / `String toUpperCase()`: Returns a new string with the case changed.
- `boolean equalsIgnoreCase(String anotherString)`: Compares two strings, ignoring case differences.
- `boolean startsWith(String prefix)` / `boolean endsWith(String suffix)`: Checks for prefix or suffix.
- `String replace(char oldChar, char newChar)`: Replaces all occurrences of a character.
- `String trim()`: Returns a new string with leading and trailing whitespace removed. It does not affect whitespace in the middle of the string.

3 String Concatenation: `+` vs. `concat()`

There are two primary ways to join strings, and the exam tests their subtle differences.

- **The `+` Operator:** This is the most common way. The Java compiler is smart and often optimizes this by using `StringBuilder` behind the scenes. Its most important rule is that if *any* operand in a '+' expression is a 'String', all other operands are converted to strings. Pay close attention to the order of operations.

```
// Numbers are added first, then converted to String
System.out.println(1 + 2 + "a"); // Prints "3a"

// "a" makes the whole expression a String concatenation
System.out.println("a" + 1 + 2); // Prints "a12"
```

- **The `concat()` Method:** This method is stricter. It only accepts a `String` as an argument and is called on a `String` object. A key difference tested on the exam is handling of `null`.

```
String s = "x";
System.out.println(s + null); // Prints "xnull"

s.concat(null); // Throws a NullPointerException
```

4 Mutable Alternatives: `StringBuilder` and `StringBuffer`

When you need to perform many string modifications (e.g., building a string in a loop), using immutable `String`s is very inefficient, as it creates a new object for every change. For these scenarios, use a mutable alternative:

- `StringBuilder`: The **preferred** choice for mutable strings. It is fast because its methods are not synchronized. Use it when you are in a single-threaded context (which is most of the time).
- `StringBuffer`: An older, thread-safe version. All its modification methods (like `append`, `insert`, `delete`) are `synchronized`. This makes it slower, and it should only be used if you need to modify a string from multiple threads.

For the exam, remember: **StringBuilder is fast and not thread-safe; StringBuffer is slow and thread-safe**.

5 Key Takeaways for the 1Z0-808 Exam

- **Immutability First:** Always think, "This method returns a NEW string." Don't fall for code that seems to modify a string in place.
- **Pool vs. Heap:** Understand `"literal"` vs. `new String("literal")` and its implication for the `==` operator.
- **Concatenation Rules:** Know the order of operations for the `+` operator and the difference in `null` handling between `+` and `concat()`.
- **Mutable Means `StringBuilder`:** If the question involves building a string through multiple modifications, the efficient answer is `StringBuilder`.