When two object references point to the same StringBuilder instance, how are changes propagated?

by Simon Roberts and Mikalai Zaikin

January 10, 2020

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. The “intermediate” and “advanced” designations refer to the exams, rather than to the questions, although in almost all cases, “advanced” questions will be harder. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

Given the class:

```java
class TextProcessor {
    public static void main(String[] args) {
        StringBuilder originalText = new StringBuilder("ABC");
        originalText.append("DEF");
        originalText = edit(text);
        System.out.println(originalText);
    }
    static StringBuilder edit(StringBuilder originalText) {
        return originalText.delete(2, 3);
    }
}
```

What is the output? Choose one.

A. ABCDEFXYZ
B. ABDEFXYZ
C. ABEFXYZ
D. ABXYZ

Answer. This code investigates mutability in the StringBuilder and the implications of multiple references to the same object.

First, the code creates a StringBuilder containing "ABC" and assigns it to the originalText variable.

Next, another variable, text, is declared; it points to the same StringBuilder("ABC") object on the heap. At this point, there are two variables that contain the same reference value. (The hypothetical value of 0x028C25B0EAA0 is used in Figure 1, but whatever the value might actually be is of no interest, nor is it generally accessible to the Java programmer.) Therefore, both are pointing to the same object on the heap.
Figure 1. originalText and text are reference values. They contain the value 0x028C25B0EAA0, which in this example is the location in memory where the StringBuilder instance is located.

Then the append("DEF") operation is done. This causes two things:

- It modifies the original StringBuilder object.
- Both variables keep pointing to the same object.

Note that the effect of modifying the original StringBuilder is very different from the behavior of a String. In the case of the String type, the original object cannot be modified; instead, a new object is created to represent the change, while the original is unmodified.

At this point, you have both text and originalText pointing to the same StringBuilder with "ABCDEF" content. Therefore, even though the change was performed using the originalText variable, the effect is to change the one object that both variables "see." At this point, the situation has changed to look like Figure 2:

Figure 2.

Then the edit method is invoked. This method modifies the object referred to by its argument and also returns the reference to that object as its result. The edit method body uses the delete method of the StringBuilder. The documentation describes StringBuilder.delete(start, end) as follows:

Removes the characters in a substring of this sequence. The substring begins at the specified start and extends to the character at index end – 1 or to the end of the sequence if no such character exists. If start is equal to end, no changes are made.

Remember that like most things in Java, indexes in a StringBuilder start from zero. This tells us that the edit method deletes a single character—that is, the character sequence starting at index 2 but stopping before index 3. In other words, it removes the letter c from the text. The value it returns is the original object reference again—in this example, 0x028C25B0EAA0. Therefore, the assignment to the originalText variable doesn't change anything. Now the picture looks like Figure 3:
The last change is that "XYZ" is appended to the variable text. This of course is identical to appending it to originalText, because both still refer to the same StringBuilder object. As with the delete method, the append method returns the reference to the original object. In this example, that is still 0x028C25B0EAA0, and as with the edit operation, the assignment of this value to originalText again changes nothing. Now the situation looks like Figure 4:

Finally, the code prints the value seen from originalText, which you can see is ABDEFXY2 (with no C in the middle). The value was overwritten with the same value again. Therefore, there is still no change. This means that option B is correct and options A, C, and D are incorrect.

Let's recap the key elements that underpin this result:

- A single StringBuilder instance exists throughout execution.
- Both variables point to the same single object on the heap, even when they are reassigned.
- Objects of the StringBuilder class are mutable; operations on them change the contents of that original object. This is fundamentally different from the behavior of a String, where new objects are created and the original ones can never be changed.
- Mutator operations return references to the same instance of the StringBuilder.

You might wonder: What's the use of having mutation methods return the reference to the object they just mutated? This behavior allows a coding style called method chaining. Because each mutating method returns this, it's possible to write code of this form:

```java
StringBuilder sb = new StringBuilder("ABC")
    .append("def")
    .delete(1,3);
```

This code would result in the object referred to by sb containing "Adef". One particular design pattern that uses this style is the “builder” pattern,
which is typically used for creating objects that can be initialized in many different ways. The builder avoids the need to create many variations of a constructor or factory where each has a different list of parameters. Instead, the proposed configuration elements are specified by chained calls to initialization methods in a manner similar to the example just given. It’s interesting to wonder whether this is how the `StringBuilder` got its name.

The correct option is B.

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun’s first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O’Reilly Safari Books Online service). He remains involved with Oracle’s Java certification projects.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

Share this Page

Facebook  Twitter  LinkedIn  Email