# String, StringBuffer, and StringBuilder Mind-Bending Challenges!

> *"Strings in Java are simple… until they're not."*

Java's `String`, `StringBuffer`, and `StringBuilder` behave differently in tricky situations. Let's explore some puzzles that will challenge your understanding.

## 1. The Immutable String Puzzle

```
public class StringPuzzle {
    public static void main(String[] args) {
        String str = "Hello";
        str.concat(" World");
        System.out.println(str);
    }
}
```

### What's the output?

```
Hello
```

✔️ **Why?** Strings are immutable in Java. The `concat()` method returns a **new String** rather than modifying the original.

### Correct Version:

```
str = str.concat(" World");
System.out.println(str); // Hello World
```

## 2. The StringBuilder Capacity Trick

```
public class CapacityDemo {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder(5);
        sb.append("12345");
        sb.append("6789");
        System.out.println(sb);
    }
}
```

### What's the output?

```
123456789
```

✔️ **Why?** Even though the initial capacity is `5`, `StringBuilder` automatically expands its size when needed.

## 3. The Reverse Surprise

```
public class ReverseDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("ABCD");
        sb.reverse();
        System.out.println(sb);
    }
}
```

### What's the output?

```
DCBA
```

✔️ **Why?** Unlike `String`, both `StringBuffer` and `StringBuilder` are **mutable**, allowing in-place reversal.

## 4. The `null` Concatenation Mystery

```
public class NullConcatDemo {
    public static void main(String[] args) {
        String str = null;
        str += " World";
        System.out.println(str);
    }
}
```

### What's the output?

```
null World
```

✔️ **Why?** When `null` is concatenated with another string, Java treats `null` as the string literal "null".

## 5. The `StringBuffer` Deadlock Danger

```
 class SyncDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Start");
        new Thread(() -> {
            synchronized (sb) {
                for (int i = 0; i < 5; i++) {
                    sb.append(" A");
                    System.out.println(sb);
                    try { Thread.sleep(100); } catch (InterruptedException
e) {}
                }
            }
        }).start();
        new Thread(() -> {
            synchronized (sb) {
                for (int i = 0; i < 5; i++) {
                    sb.append(" B");
                    System.out.println(sb);
                    try { Thread.sleep(100); } catch (InterruptedException
e) {}
                }
            }
        }).start();
    }
}
```

### What's the output?

```
 Start A
Start A A
Start A A A
Start A A A A
Start A A A A A
Start A A A A A B
Start A A A A A B B
```

✔️ **Why?** `StringBuffer` is thread-safe due to its synchronized methods, but if both threads hold the same object lock, they must wait for each other, making deadlocks possible.

## Final Thoughts

Mastering `String`, `StringBuffer`, and `StringBuilder` is essential for writing efficient and bug-free Java code. These tricky behaviors often appear in coding interviews and performance challenges.