Why Java Strings Are Immutable?

Immutable strings are inherently thread-safe, as their values cannot be changed, making them safe to use across multiple threads. Immutable strings enhance security by preventing unauthorized changes to string values, which is crucial for sensitive data like passwords and network connections. It allows for the reuse of string literals, reducing memory overhead and improving performance. Hashcodes of immutable strings can be cached, making operations like hash-based lookups faster. Immutable objects are more predictable and reliable, reducing bugs and simplifying debugging.

StringBuffer Class in Java

`StringBuffer` objects are mutable, meaning their values can be modified after creation. `StringBuffer` methods are synchronized, making it thread-safe for use in concurrent applications. More efficient for concatenation and modification operations compared to `String` due to its mutability. It includes methods like `append()`, `insert()`, `delete()`, `reverse()`, and `setCharAt()`. It suitable for scenarios where strings undergo frequent modifications, and thread safety is required.

Syntax:

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
```

Example:

```
public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.append(" World");
        System.out.println(sb);
    }
}
```

StringBuilder Class in Java

`StringBuilder` is similar to `StringBuffer.` `StringBuilder` objects are mutable, allowing for modification after creation. Unlike `StringBuffer`, `StringBuilder` is not synchronized, making it faster but not thread-safe. More efficient for string modifications compared to `String` due to its mutable nature. It includes methods like `append()`, `insert()`, `delete()`, `reverse()`, and `setCharAt()`. It suitable for scenarios where strings undergo frequent modifications, and thread safety is not a concern.

Syntax:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
```

Example:

```
public class StringBuilderExample {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
        sb.append(" World");
        System.out.println(sb);
    }
}
```

StringTokenizer Class in Java

`StringTokenizer` is used to break a string into tokens based on specified delimiters. It is a part of the `java.util` package and considered a legacy class, with `split()` method from `String` class often preferred. It provides constructors to specify the string to be tokenized and optionally, the delimiters. It includes methods like `hasMoreTokens()`, `nextToken()`, and `countTokens()`. It is useful for simple tokenization tasks, but less flexible and efficient compared to the `split()` method.

Syntax:

```
StringTokenizer st = new StringTokenizer("Hello World");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Example:

```
import java.util.StringTokenizer;

public class StringTokenizerExample {
    public static void main(String[] args) {
        StringTokenizer st = new StringTokenizer("Hello World");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

StringJoiner Class in Java

`StringJoiner` is used to construct a sequence of characters separated by a delimiter. It allows specifying a delimiter, and optionally, a prefix and suffix for the resulting string. Unlike `StringJoiner` is mutable, allowing for efficient concatenation. It includes methods like `add()`, `toString()`, `length()`, and `merge()`. It suitable for scenarios where multiple strings need to be concatenated with specific delimiters, prefixes, and suffixes.

Syntax:

```
StringJoiner sj = new StringJoiner(", ");
```

```
sj.add("Hello");
sj.add("World");
```

Example:

```
import java.util.StringJoiner;

public class StringJoinerExample {
    public static void main(String[] args) {
        StringJoiner sj = new StringJoiner(", ");
        sj.add("Hello");
        sj.add("World");
        System.out.println(sj);
    }
}
```