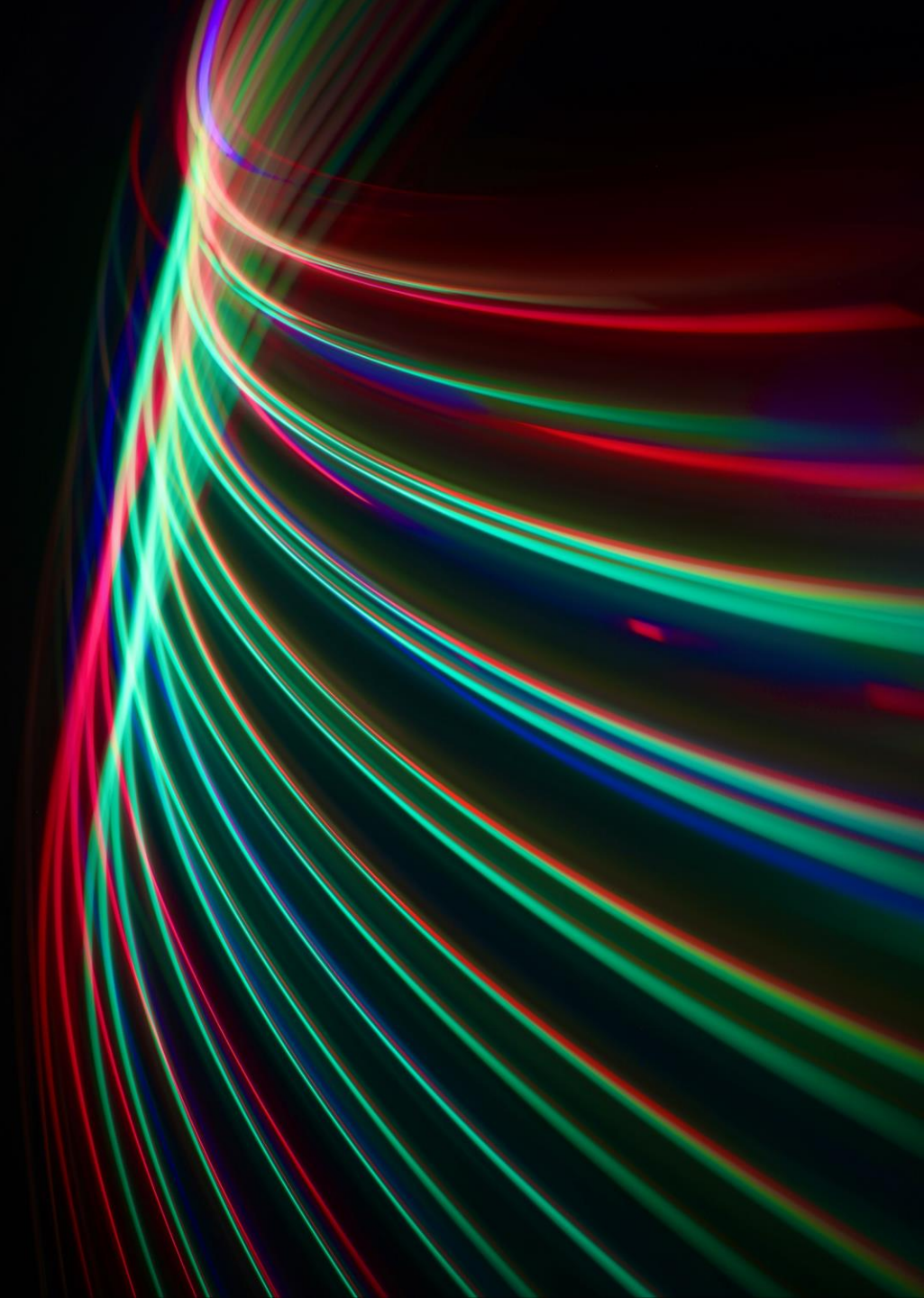


String in Java

String, StringBuilder, StringBuffer

20210463 박연종

2024.06.12.



목차

- 1 String
- 2 String을 불변 객체로 설계한 이유
- 3 쓸모 없는 메모리 공간이 늘어나요
- 4 StringBuffer, 기존 공간을 재사용해요
- 5 싱글 스레드에서도 동기화해요
- 6 StringBuilder, 동기화 없이 기존 공간을 재사용해요
- 7 싱글 스레드와 멀티 스레드에서의 StringBuffer, StringBuilder
- 8 이어 붙이기 성능 비교

불변 객체, String

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
    Constable, ConstantDesc {
```

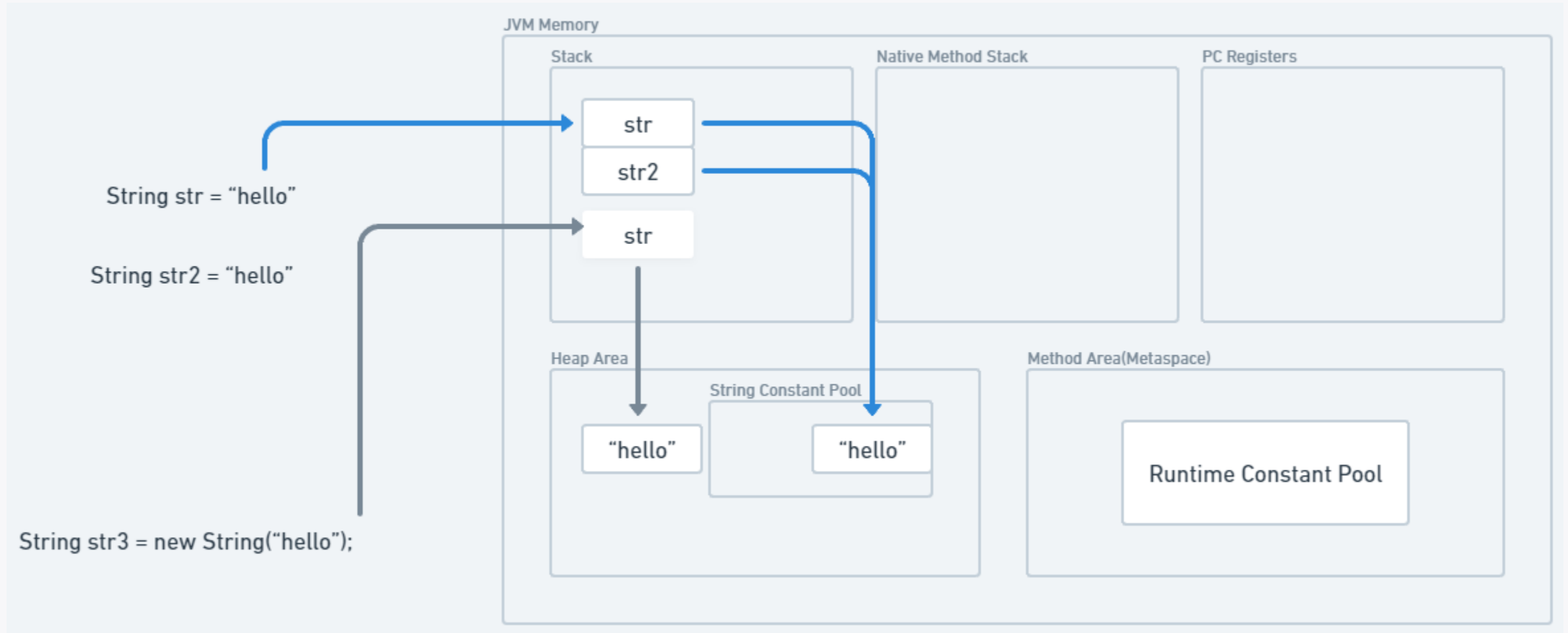
The value is used for character storage.

Implementation This field is trusted by the VM, and is a subject to constant folding if String

Note: instance is constant. Overwriting this field after construction will cause problems. Additionally, it is marked with `Stable` to trust the contents of the array. No other facility in JDK provides this functionality (yet). `Stable` is safe here, because value is never null.

```
    @Stable
    private final byte[] value;
```

불변 객체로 설계한 이유: 캐싱과 String Constant Pool



불변 객체로 설계한 이유: 캐싱과 String Constant Pool

```
1 public class Main {  
2     public static void main(String[] args) {  
3         String str = "hello";  
4         String str2 = "hello";  
5         String str3 = new String(original: "hello");  
6           
7         System.out.println("str hashCode: " + Integer.toHexString(System.identityHashCode(str)));  
8         System.out.println("str2 hashCode: " + Integer.toHexString(System.identityHashCode(str2)));  
9         System.out.println("str == str2 : " + (str == str2));  
10        System.out.println("str3 hashCode: " + Integer.toHexString(System.identityHashCode(str3)));  
11        System.out.println("str == str3 : " + (str == str3));  
12    }  
13 }
```

```
C:\Users\calva\.jdk\openjdk-22.0.  
str hashCode: 10f87f48  
str2 hashCode: 10f87f48  
str == str2 : true  
str3 hashCode: 5b6f7412  
str == str3 : false  
  
Process finished with exit code 0
```

새로운 문자열 객체를 만드는데
걸리는 시간을 절약

불변 객체로 설계한 이유: 보안

```
void criticalMethod(String userName) { no usages
    // 파라미터로 넘어온 문자열에 대한 검증 절차
    if (!isAlphaNumeric(userName)) {
        throw new SecurityException();
    }

    // 문자열이 넘어온 후 어느 정도의 시간이 지남
    initializeDatabase();

    // 이후 DB에 SQL 쿼리를 날림
    // 이때 불변이 아니면 어떤 프로그램에서 파라미터로 넘어온 문자열을
    // 수정했을 수도 있는 불안함이 있음
    connection.executeUpdate("UPDATE Customers SET Status = 'Active' " +
        "WHERE UserName = '" + userName + "'");
}
```

불변 객체로 설계한 이유

캐싱과 String Constant Pool

String Constant Pool에 리터럴 문자열 하나만 저장해 재사용함으로써 힙 영역의 메모리 공간을 절약할 수 있음

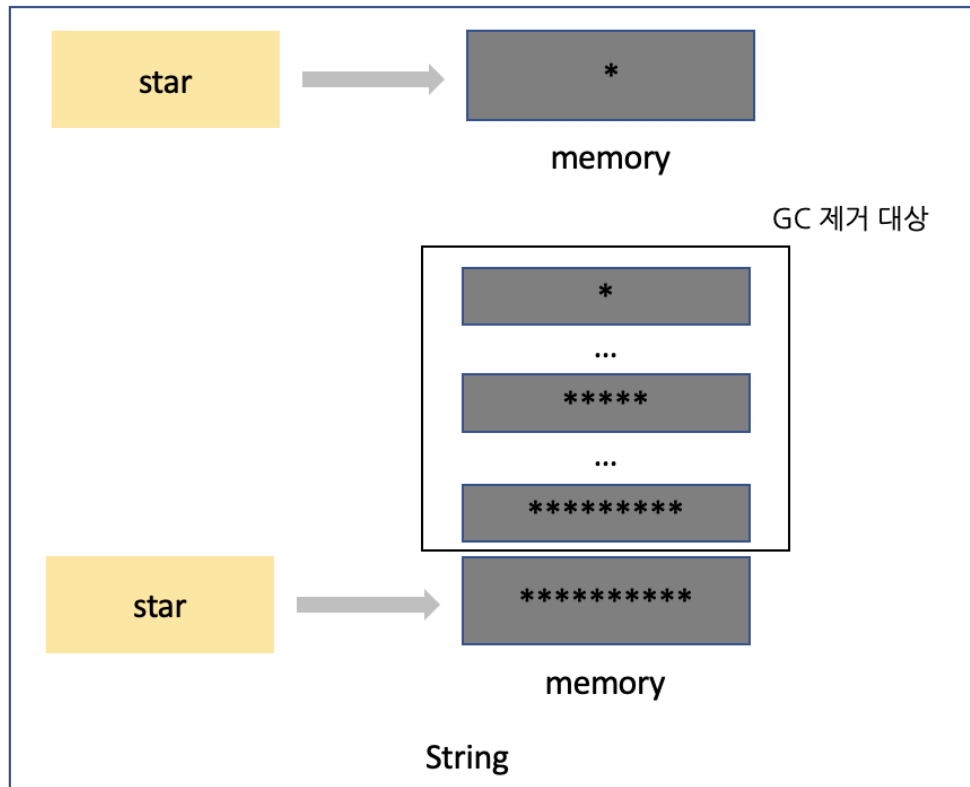
보안

파라미터의 인자로 문자열을 전달할 때 검증 등의 시간이 소요되는 과정을 거친 이후 누군가 문자열을 수정했을 수도 있다는 불안함이 존재

스레드 안전

문자열이 변하지 않으므로 동시에 실행되는 여러 스레드에서 안정적으로 공유 가능

쓸모 없는 메모리 공간이 늘어나요

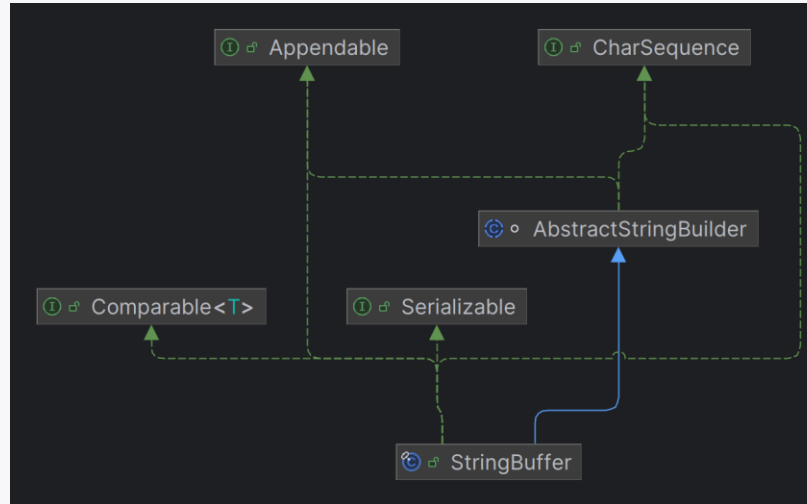


```
1 public class Main {  
2     public static void main(String[] args) {  
3         String star = "*";  
4         System.out.println(Integer.toHexString(System.identityHashCode(star)));  
5         for (int i = 1; i < 10; i++) {  
6             star += "*";  
7             System.out.println(Integer.toHexString(System.identityHashCode(star)));  
8         }  
9     }  
10 }  
11 }
```

```
C:\Users\calva\.jdk\openjdk-22.0.  
10f87f48  
1d81eb93  
7291c18f  
34a245ab  
7cc355be  
6e8cf4c6  
12edcd21  
34c45dca  
52cc8049  
5b6f7412  
  
Process finished with exit code 0
```

- 불변 객체이므로 매번 새로운 String 객체를 만들어 메모리 사용량이 증가함
- String 객체 생성과 Garbage Collector의 동작에 시간이 소요됨

StringBuffer



```
abstract sealed class AbstractStringBuilder implements Appendable, CharSequence
```

```
    permits StringBuilder, StringBuffer {
```

```
        The value is used for character storage.
```

```
        byte[] value;
```

```
    public synchronized StringBuffer append(StringBuffer sb) {
```

```
        toStringCache = null;
```

```
        super.append(sb);
```

```
        return this;
```

```
    }
```

StringBuffer 내부 문자열의 길이

Constructs an `AbstractStringBuilder` that contains the same characters as the specified `CharSequence`. The initial capacity of the string builder is `16` plus the length of the `CharSequence` argument.

The contents are unspecified if the `CharSequence` is modified during string construction.

Params: `seq` – the sequence to copy.

```
AbstractStringBuilder(CharSequence seq) {  
    int length = seq.length();  
    if (length < 0) {  
        throw new NegativeArraySizeException("Negative length: " + length);  
    }  
    int capacity = (length < Integer.MAX_VALUE - 16)  
        ? length + 16 : Integer.MAX_VALUE;  
  
    final byte initCoder;  
    if (COMPACT_STRINGS) {  
        if (seq instanceof AbstractStringBuilder asb) {  
            initCoder = asb.getCoder();  
            maybeLatin1 |= asb.maybeLatin1;  
        } else if (seq instanceof String s) {  
            initCoder = s.coder();  
        } else {  
            initCoder = LATIN1;  
        }  
    } else {  
        initCoder = UTF16;  
    }  
  
    coder = initCoder;  
    value = (initCoder == LATIN1)  
        ? new byte[capacity] : StringUTF16.newBytesFor(capacity);  
    append(seq);  
}
```

일반적인 문자열로 초기화 시
현재 문자열의 길이에 16을 더한 값을 반환

StringBuffer 내부 문자열 추가 시 길이 증가

```
public AbstractStringBuilder append(String str) {  
    if (str == null) {  
        return appendNull();  
    }  
    int len = str.length();  
    ensureCapacityInternal( minimumCapacity: count + len);  
    putStringAt(count, str);  
    count += len;  
    return this;  
}
```

```
private void ensureCapacityInternal(int minimumCapacity) {  
    // overflow-conscious code  
    int oldCapacity = value.length >> coder;  
    if (minimumCapacity - oldCapacity > 0) {  
        value = Arrays.copyOf(value,  
            newLength: newCapacity(minimumCapacity) << coder);  
    }  
}
```

기존 공간이 기존 문자열 + 추가 문자열 길이보다 크면 새 배열 생성

```
private int newCapacity(int minCapacity) {  
    int oldLength = value.length;  
    int newLength = minCapacity << coder;  
    int growth = newLength - oldLength;  
    int length = ArraysSupport.newLength(oldLength, growth, prefGrowth: oldLength + (2 << coder));  
    if (length == Integer.MAX_VALUE) {  
        throw new OutOfMemoryError(s: "Required length exceeds implementation limit");  
    }  
    return length >> coder;  
}
```

```
static final byte LATIN1 = 0;  
static final byte UTF16 = 1;
```

StringBuffer 내부 문자열 추가 시 길이 증가

기존 공간의 크기 (기존+추가 문자열 길이) - 기존 공간 크기

```
public static int newLength(int oldLength, int minGrowth, int prefGrowth) {  
    // preconditions not checked because of inlining  
    // assert oldLength >= 0  
    // assert minGrowth > 0  
    int prefLength = oldLength + Math.max(minGrowth, prefGrowth); // might overflow  
    if (0 < prefLength && prefLength <= SOFT_MAX_ARRAY_LENGTH) {  
        return prefLength;  
    } else {  
        // put code cold in a separate method  
        return hugeLength(oldLength, minGrowth);  
    }  
}
```

기존 공간 + 2

기존 공간 크기 + (기존+추가 문자열 길이) - 기존 공간 크기 =

(기존 공간 크기 + 기존 공간 크기 + 2) 또는 (기존 문자열 길이 + 추가 문자열 길이)

StringBuffer 내부 문자열 추가 시 길이 증가

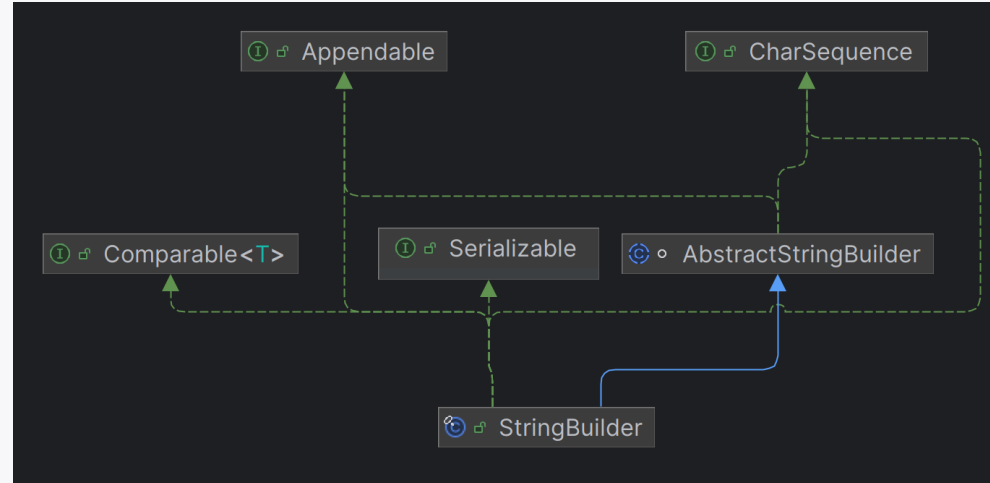
```
newLength = oldLength + appendLength
if (oldCapacity ≤ newLength) then
    // 기존 문자열 뒤에 새 문자열을 추가합니다.
else
    newCapacity = oldCapacity * 2 + 2
    if (newLength ≤ newCapacity) then
        // newCapacity 크기의 배열을 만들고, 기존 문자열을 복사한 후 새 문자열을 추가합니다.
    else
        // newLength 크기의 배열을 만들고, 기존 문자열을 복사한 후 새 문자열을 추가합니다.
```

StringBuffer 문자열 길이 확인

```
3 public class Main {
4     public static void main(String[] args) {
5         // 초기화 시 문자열이 저장되는 공간의 크기는 문자열 길이에 16을 더한 값
6         StringBuffer sb1 = new StringBuffer();
7         System.out.println("sb1 초기 공간 = " + sb1.capacity());
8         // 초기화 시 문자열이 저장되는 공간의 크기는 문자열 길이에 16을 더한 값
9         StringBuffer sb2 = new StringBuffer("1");
10        System.out.println("sb2 초기 공간 = " + sb2.capacity());
11        // 초기화 시 문자열이 저장되는 공간의 크기는 문자열 길이에 16을 더한 값
12        System.out.println("::::: 기존 공간을 재사용해요 :::::");
13        System.out.println("기대 = " + 16);
14        sb1.append("1234567890123456"); // now: 16
15        System.out.println("실제 = " + sb1.capacity());
16        // 초기화 시 문자열이 저장되는 공간의 크기는 문자열 길이에 16을 더한 값
17        System.out.println("::::: 기존 공간보다 크면, oldCapacity*2+2 :::::");
18        System.out.println("기대 = " + (16 * 2 + 2));
19        sb1.append("1234567890123456"); // now: 16 + 16
20        System.out.println("실제 = " + sb1.capacity()); // 34
21        // 초기화 시 문자열이 저장되는 공간의 크기는 문자열 길이에 16을 더한 값
22        // 70-32 = 38
23        System.out.println("::::: 기존 공간보다 크면, oldCapacity*2+2보다 크면 newLength :::::");
24        System.out.println("기대 = " + (16 + 16 + 44));
25        sb1.append("12345678901234567890123456789012345678901234"); // now: 16 + 16 + 44
26        System.out.println("실제 = " + sb1.capacity()); // 76
27    }
28 }
```

```
sb1 초기 공간 = 16
sb2 초기 공간 = 17
::::: 기존 공간을 재사용해요 :::::
기대 = 16
실제 = 16
::::: 기존 공간보다 크면, oldCapacity*2+2 :::::
기대 = 34
실제 = 34
::::: 기존 공간보다 크면, oldCapacity*2+2보다 크면 newLength :::::
기대 = 76
실제 = 76
```

StringBuilder



```
abstract sealed class AbstractStringBuilder implements Appendable, CharSequence
```

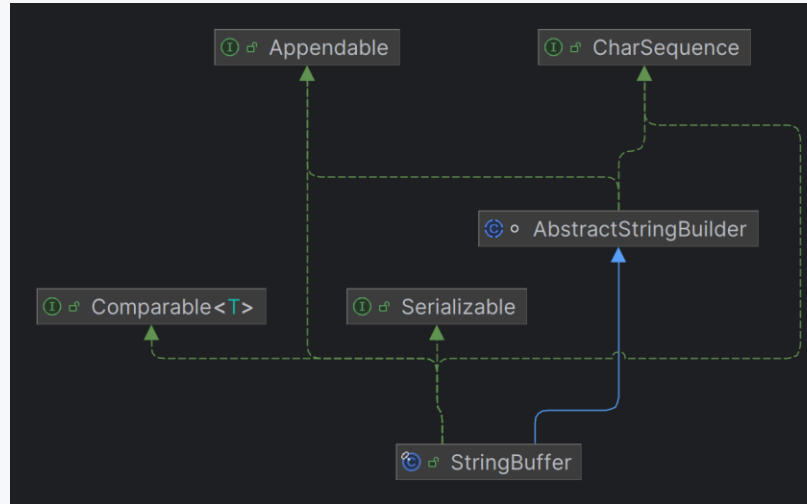
```
    permits StringBuilder, StringBuffer {
```

```
        The value is used for character storage.
```

```
        byte[] value;
```

```
    public StringBuilder append(StringBuffer sb) {
        super.append(sb);
        return this;
    }
```

StringBuilder와 StringBuffer



```
abstract sealed class AbstractStringBuilder implements Appendable, CharSequence
```

```
    permits StringBuilder, StringBuffer {
```

```
        The value is used for character storage.
```

```
        byte[] value;
```

```
    public synchronized StringBuffer append(StringBuffer sb) {
```

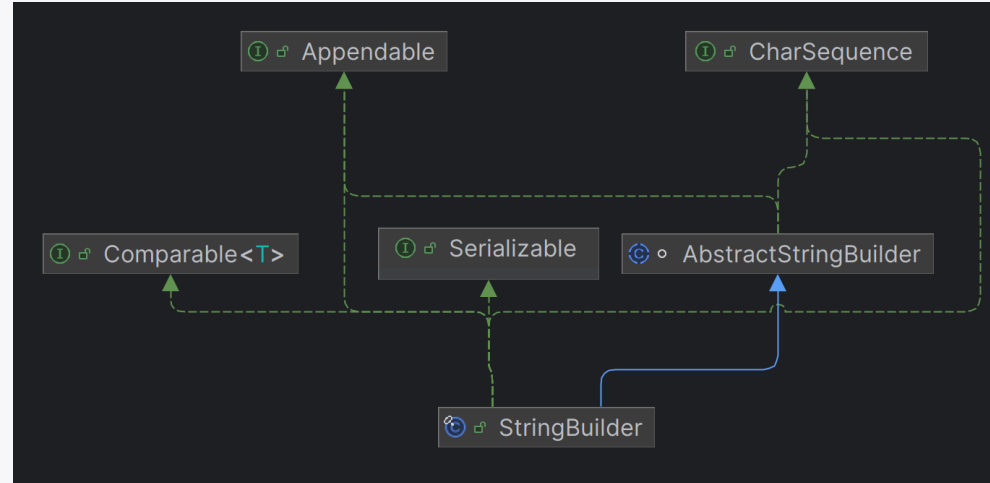
```
        toStringCache = null;
```

```
        super.append(sb);
```

```
        return this;
```

```
    }
```


StringBuilder



```
abstract sealed class AbstractStringBuilder implements Appendable, CharSequence
```

```
    permits StringBuilder, StringBuffer {
```

```
        The value is used for character storage.
```

```
        byte[] value;
```

```
    public StringBuilder append(StringBuffer sb) {
        super.append(sb);
        return this;
    }
```

synchronized 외에는 다른 점이 없고 둘 모두 `AbstractStringBuilder`를 상속 받았음을 확인할 수 있음

StringBuffer와 StringBuilder, 싱글 스레드 비교

```
3 ▶ public class Main extends Thread{
4 ▶     public static void main(String[] args) {
5         StringBuffer stringBuffer = new StringBuffer();
6         StringBuilder stringBuilder = new StringBuilder();
7
8         long startTime1 = System.nanoTime();
9         for(int i=0; i<20000; i++){
10             stringBuffer.append("1");
11         }
12         long endTime1 = System.nanoTime();
13
14         long startTime2 = System.nanoTime();
15         for(int i=0; i<20000; i++){
16             stringBuilder.append("1");
17         }
18         long endTime2 = System.nanoTime();
19
20         long duration1 = endTime1 - startTime1;
21         long duration2 = endTime2 - startTime2;
22         System.out.println("StringBuffer의 append()을 이용한 경우 : "+ duration1 + "ns");
23         System.out.println("StringBuilder의 append()을 이용한 경우 : "+ duration2 + "ns");
24     }
25 }
```

StringBuffer의 append()을 이용한 경우 : 2038300ns
StringBuilder의 append()을 이용한 경우 : 1711400ns

StringBuffer의 append()을 이용한 경우 : 2101400ns
StringBuilder의 append()을 이용한 경우 : 1642200ns

StringBuffer의 append()을 이용한 경우 : 2422000ns
StringBuilder의 append()을 이용한 경우 : 2352200ns

StringBuffer의 append()을 이용한 경우 : 2220000ns
StringBuilder의 append()을 이용한 경우 : 1819900ns

StringBuffer의 append()을 이용한 경우 : 1894600ns
StringBuilder의 append()을 이용한 경우 : 1699600ns

StringBuilder가 더 빠름

StringBuffer.length: 20000
StringBuilder.length: 20000

StringBuffer와 StringBuilder, 멀티 스레드 비교

```
3 public class Main extends Thread{
4     public static void main(String[] args) {
5         StringBuffer stringBuffer = new StringBuffer();
6         StringBuilder stringBuilder = new StringBuilder();
7
8         new Thread() -> {
9             for(int i=0; i<10000; i++){
10                 stringBuffer.append("1");
11                 stringBuilder.append("1");
12             }
13         }).start();
14
15         new Thread() -> {
16             for(int i=0; i<10000; i++){
17                 stringBuffer.append("1");
18                 stringBuilder.append("1");
19             }
20         }).start();
21
22         new Thread() -> {
23             try {
24                 Thread.sleep(2000);
25
26                 System.out.println("StringBuffer.length: " + stringBuffer.length());
27                 System.out.println("StringBuilder.length: " + stringBuilder.length());
28             } catch (InterruptedException e) {
29                 e.printStackTrace();
30             }
31         }).start();
32     }
33 }
```



```
C:\Users\calva\.jdk\openjdk-22.0
StringBuffer.length: 20000
StringBuilder.length: 19982

Process finished with exit code 0
```

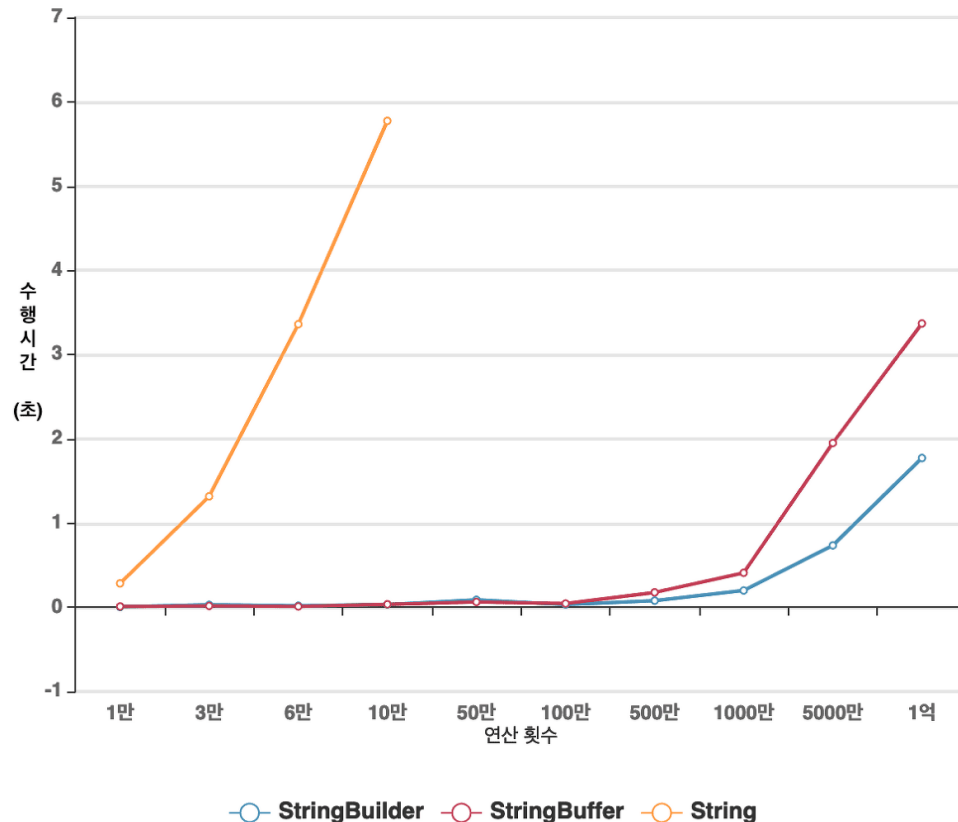
2만 번의 문자열 추가

StringBuilder의 append 메서드가
메모리 공간에 동시 접근해
문자열이 두 번 추가돼야 하나
한 번만 추가됨

String, StringBuffer, StringBuilder 문자열 추가 성능

 OpenJDK Platform binary		13.0%	1,085.4MB	0MB/s	0Mbps
 OpenJDK Platform binary		0%	104.6MB	0MB/s	0Mbps

성능 비교



String의 +연산을 이용한 경우 : 6672899ms
StringBuffer의 append()을 이용한 경우 : 384ms
StringBuilder의 append()을 이용한 경우 : 59ms

String, StringBuffer, StringBuilder

	String	StringBuffer	StringBuilder
가변 여부	불변	가변	가변
스레드 안전	O	O	X
연산 속도	느림	빠름	아주 빠름
사용 시점	적은 문자열 추가 연산	많은 문자열 추가 연산	많은 문자열 추가 연산 싱글 스레드 환경
존재	JDK 1.0부터	JDK 1.0부터	JDK 1.5부터

감사합니다

Email / park@duck.com

Insta / [@yeonjong.park](https://www.instagram.com/yeonjong.park)

GitHub / [patulus](https://github.com/patulus)

