Java – Detailed Notes on StringBuilder & StringBuffer

Introduction:

Java provides three main classes to handle strings:

- String – Immutable
- StringBuilder – Mutable, non-synchronized (faster)
- StringBuffer – Mutable, synchronized (thread-safe)

Both `StringBuilder` and `StringBuffer` are used to **manipulate strings** without creating new objects (unlike `String`).

String vs StringBuilder vs StringBuffer:

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread-safe	No	No	Yes
Performance	Slow	Fast (single-thread)	Slower (multi-thread)
Package	java.lang	java.lang	java.lang
Introduced in	Java 1.0	Java 1.5	Java 1.0

Why Use StringBuilder or StringBuffer?

- Performance: Frequent string changes using + or concat() are costly.
- Efficiency: They use a dynamic character array internally.
- Use Cases: Loops, string reversal, file parsing, log building, etc.

Constructors:

StringBuilder sb = new StringBuilder(); // default capacity 16

StringBuilder sb = new StringBuilder("Hello");

StringBuilder sb = new StringBuilder(50); // custom capacity

Same constructors exist for StringBuffer.

Common Methods:

Method	Description	Example
append(String s)	Adds string at end	sb.append("Java")
insert(int index, String s)	Inserts string at index	sb.insert(2, "Test")
replace(int start, int end, String s)	Replaces chars from start to end	sb.replace(0, 4, "Hi")
delete(int start, int end)	Deletes from start to end	sb.delete(0, 3)
reverse()	Reverses characters	sb.reverse()
toString()	Converts to String	String s = sb.toString()
capacity()	Returns capacity	sb.capacity()
charAt(int index)	Returns char at index	sb.charAt(1)
length()	Returns length of content	sb.length()

Internal Working:

- Both use char[] array internally.
- Default capacity = 16; increases as needed.
- Capacity growth formula:

newCapacity = (oldCapacity * 2) + 2;

Thread Safety:

Case	Use
Single-threaded program	Use StringBuilder for speed
Multi-threaded program	Use StringBuffer to avoid data inconsistency

Example Code – Using StringBuilder:

```
public class BuilderExample {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
        sb.append(" Java");
        sb.insert(5, ",");
        sb.replace(6, 10, "World");
        sb.reverse();
        System.out.println(sb); // Output: dlroW ,olleH
    }
}
```

StringBuffer Example (Thread-Safe):

```
public class BufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Sync");
        sb.append(" Java");
        System.out.println(sb); // Sync Java
    }
}
```

Performance Comparison:

```
public class PerformanceTest {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 100000; i++) {
            sb.append("a");
        }
        long end = System.currentTimeMillis();
        System.out.println("Time: " + (end - start));
    }
}
```

- StringBuilder ≈ Fastest
- StringBuffer ≈ Slightly slower due to synchronization
- String ≈ Very slow due to immutability

Interview Question Highlights:

Why is StringBuilder faster than StringBuffer?

Because StringBuilder doesn't use synchronization, making it faster in single-threaded scenarios.

What is the default capacity of StringBuilder?

The default capacity is 16 characters.
If you pass a string "Hello" (length 5), the total capacity becomes:
16 + 5 = 21.

Can we convert StringBuilder to String? How?

Yes. Use the .toString() method.

Can you override methods of StringBuilder?

No. StringBuilder is a final class, so it cannot be extended or overridden.

What happens if you exceed StringBuilder capacity?

It automatically increases capacity using the formula (oldCapacity * 2) + 2. It creates a new array, copies old data, and continues.

Best Practices:

- Use StringBuilder when thread safety is not needed.
- Avoid using + in loops; use StringBuilder.append().
- For thread-safe operations with multiple threads, prefer StringBuffer.
- Always check capacity in memory-sensitive applications.