String

The first topic in the Java Basics category: `String`'s IMMUTABLE design, the string pool and the `==` vs `equals()` trap, the O(n^2) cost of `+` concatenation vs. `StringBuilder`'s amortized O(1) (with a real, warmed-up measurement), `String.format()`/text blocks (Java 15+), and helper methods like split/join/replace/trim/strip.

Beginner 20 min

Last updated: October 20, 2026
Read online: https://www.learnforgex.com/en/topics/string

String

`String` is the first topic in the Java Basics category -- probably the very first class every Java programmer runs into. It looks simple, but underneath it hides an IMMUTABLE design, a special memory optimization called the "string pool", and the difference between `==` and `equals()`, which is one of the most common traps for beginners.

What Is String?

`String`, defined in the `java.lang` package, is a class that represents a sequence of characters. In Java, strings are NOT a primitive type like `char[]` -- they are full-fledged OBJECTS -- but the language makes it possible to use them almost like a primitive type, with syntax like `String s = "hello";`. Its most critical property: `String` objects are IMMUTABLE -- once created, their content never changes; every method that looks like it "modifies" the string (`toUpperCase()`, `substring()`, `replace()`, etc.) actually returns a NEW `String` object.

Why Does It Exist?

Working with text -- reading user input, building a file path, printing an error message -- is a basic need of almost every program. `String` being immutable is a deliberate design decision: immutable objects are THREAD-SAFE (multiple threads can safely share the same `String`, since no one can change it), its `hashCode()` value can be computed once and cached (which is why using `String` as a `HashMap` key is common and fast), and the JVM can share identical text through a "string pool" instead of storing it as separate objects over and over (saving memory).

History

The `String` class has been part of Java since version 1.0 (1996) -- one of the oldest, most fundamental pieces of the language. The string pool (intern table) concept has also existed from the start. Java 7 (2011) changed `substring()`'s old implementation: previously, a `substring()` call shared the ORIGINAL character array (saving memory but carrying a hidden memory-leak risk -- a small substring could keep a huge original array alive in memory); from Java 7 onward, every `substring()` creates its own independent copy. Java 9 (2017) introduced "Compact Strings", storing strings that only contain Latin-1 (single-byte) characters as a `byte[]` instead of a `char[]` internally, reducing memory usage. Java 11 (2018) added methods like `strip()` / `isBlank()` / `repeat()`, and Java 15 (2020) added text blocks (`"""`) and the `formatted()` method.

Basic Usage and Immutability

The most common way to create a `String` is writing a LITERAL (like `"hello"`). Basic inspection methods include `length()`, `charAt()`, `substring()`, `indexOf()`, `contains()`. The critical point: calling a method like `toUpperCase()` does NOT modify the original `String` -- it returns a new `String`, and the original stays the same.

```
public class StringBasicsExample {
    public static void main(String[] args) {
        // A String literal -- Java interns literals in the "string pool" automatically.
        String greeting = "Hello, World!";

        // Basic inspection methods.
        System.out.println("Value: " + greeting);
        System.out.println("length(): " + greeting.length());
        System.out.println("charAt(7): " + greeting.charAt(7));
        System.out.println("isEmpty(): " + greeting.isEmpty());
        System.out.println("isBlank(): " + "   ".isBlank());

        // Strings are IMMUTABLE -- every "modifying" method returns a NEW String,
        // it never changes the original.
        String upper = greeting.toUpperCase();
        System.out.println("Original after toUpperCase(): " + greeting);
        System.out.println("New uppercase String: " + upper);
        System.out.println("Same object? " + (greeting == upper));

        // substring(), indexOf(), contains().
        System.out.println("substring(7): " + greeting.substring(7));
        System.out.println("substring(0, 5): " + greeting.substring(0, 5));
        System.out.println("indexOf(\"World\"): " + greeting.indexOf("World"));
        System.out.println("contains(\"World\"): " + greeting.contains("World"));

        // Concatenation with + creates yet another new String.
        String combined = greeting + " Nice to meet you.";
        System.out.println("Concatenated: " + combined);
    }
}
```

Expected Output

```
Value: Hello, World!
length(): 13
charAt(7): W
isEmpty(): false
isBlank(): true
Original after toUpperCase(): Hello, World!
New uppercase String: HELLO, WORLD!
Same object? false
substring(7): World!
substring(0, 5): Hello
indexOf("World"): 7
contains("World"): true
Concatenated: Hello, World! Nice to meet you.
```

If you don't ASSIGN a `String` method's return value to a variable, the call has no effect at all (`greeting.toUpperCase();` on its own does nothing) -- this is one of the most common beginner mistakes related to immutability.

The String Pool and == vs equals()

Java shares string LITERALS that have the same text through a "string pool" -- two separate `"hello"` literals are actually the SAME object in memory. But a string created with `new String(...)` is ALWAYS a new, separate object, even if it has the same text as a value already in the pool. The `==` operator compares object IDENTITY (is it the same memory address), while `equals()` compares CONTENT -- confusing the two leads to mysterious bugs that "sometimes work, sometimes don't".

```
public class StringPoolAndEqualityExample {
    public static void main(String[] args) {
        // Two literals with the same text are the SAME object -- the compiler/JVM
        // reuses a single entry in the "string pool" (an intern table).
        String a = "hello";
        String b = "hello";
        System.out.println("a == b (both literals): " + (a == b));

        // `new String(...)` FORCES a brand-new object on the heap, even if the
        // pool already has an identical value.
        String c = new String("hello");
        System.out.println("a == c (c is `new String(...)`): " + (a == c));
        System.out.println("a.equals(c): " + a.equals(c));

        // intern() looks up (or adds) the pool entry for this value's content,
        // so it goes back to being the same object as the literal.
        String d = c.intern();
        System.out.println("a == d (d is c.intern()): " + (a == d));

        // Compile-time constant expressions are folded into a single literal by
        // the compiler, so they ALSO land in the pool.
        String e = "hel" + "lo"; // constant-folded at compile time
        System.out.println("a == e (\"hel\" + \"lo\" is constant-folded): " + (a == e));

        // But concatenation involving a NON-constant (a variable) happens at
        // runtime and produces a new object -- it is NOT interned automatically.
        String prefix = "hel";
        String f = prefix + "lo";
        System.out.println("a == f (prefix + \"lo\", built at runtime): " + (a == f));
        System.out.println("a.equals(f): " + a.equals(f));

        System.out.println();
        System.out.println("Rule of thumb: NEVER use == to compare String content.");
        System.out.println("Always use equals() (or equalsIgnoreCase()).");
    }
}
```

Expected Output

```
a == b (both literals): true
a == c (c is `new String(...)`): false
a.equals(c): true
a == d (d is c.intern()): true
a == e ("hel" + "lo" is constant-folded): true
a == f (prefix + "lo", built at runtime): false
a.equals(f): true

Rule of thumb: NEVER use == to compare String content.
Always use equals() (or equalsIgnoreCase()).
```

> Comparing string content with `==` is a CLASSIC mistake -- in some cases (two literals) it happens to give the right answer, but with `new String(...)` or a string concatenated at runtime, it SILENTLY produces the wrong result. Always use `equals()` (or `equalsIgnoreCase()` for case-insensitive comparison) for content comparison.

String Concatenation and Performance

Concatenating strings with `+` reads comfortably, but since `String` is immutable, every `+=` call creates a NEW `String` object -- `result += "x"` repeated N times in a loop means roughly O(n²) total work (all the previous content gets copied again at every step). `StringBuilder` keeps a SINGLE mutable character array, reducing that cost to amortized O(1).

```
public class StringConcatenationPerformanceExample {
    public static void main(String[] args) {
        int rounds = 30_000;

        // Warm-up -- run both paths a lot before measuring, so the JIT compiler has
        // already optimized both loops by the time we time them.
        for (int i = 0; i < 3_000; i++) {
            concatenateWithPlus(rounds / 10);
            concatenateWithStringBuilder(rounds / 10);
        }

        long plusStart = System.nanoTime();
        String plusResult = concatenateWithPlus(rounds);
        long plusNanos = System.nanoTime() - plusStart;

        long builderStart = System.nanoTime();
        String builderResult = concatenateWithStringBuilder(rounds);
        long builderNanos = System.nanoTime() - builderStart;

        System.out.println("Building a String out of " + rounds + " pieces in a loop:");
        System.out.println("  '+' operator in a loop: " + (plusNanos / 1_000_000) + " ms");
        System.out.println("  StringBuilder.append(): " + (builderNanos / 1_000_000) + "
ms");
        System.out.println("Same final length? " + (plusResult.length() == builderResult.
length()));
    }

    // Each `+=` on a String inside a loop creates a BRAND-NEW String object (since
    // String is immutable) -- for N iterations this is roughly O(n^2) total work,
    // because every intermediate result gets copied again.
    private static String concatenateWithPlus(int rounds) {
        String result = "";
        for (int i = 0; i < rounds; i++) {
            result += "x";
        }
        return result;
    }

    // StringBuilder keeps ONE mutable internal char array and grows it as needed
    // (amortized O(1) per append) -- this is the recommended way to build a String
    // piece by piece in a loop.
    private static String concatenateWithStringBuilder(int rounds) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < rounds; i++) {
            result.append("x");
        }
        return result.toString();
    }
}
```

Real measurement (warmed up -- both paths were run thousands of times before timing): building a string out of 30,000 pieces, the `+` operator inside a loop consistently took tens of milliseconds (~63-80 ms, varying between runs), while `StringBuilder.append()` was too fast to measure at this scale (0 ms) -- confirming the expected $O(n^2)$/$O(n)$ difference with a real run.

Building Mutable Strings with StringBuilder

`StringBuilder` is `String`'s mutable sibling -- methods like `append()`, `insert()`, `replace()`, `delete()`, `reverse()` modify the SAME object in place, they don't return a new one. If you're building a string piece by piece, especially in a loop, `StringBuilder` is the right tool; once you're done, `toString()` converts it into an immutable `String`.

```
public class StringBuilderExample {
    public static void main(String[] args) {
        // Unlike String, StringBuilder is MUTABLE -- its methods change the same
        // object in place instead of returning a new one.
        StringBuilder sb = new StringBuilder("Hello");
        System.out.println("Initial: " + sb);

        sb.append(", World");
        System.out.println("After append(): " + sb);

        sb.append('!');
        System.out.println("After append(char): " + sb);

        sb.insert(5, " there");
        System.out.println("After insert(5, \" there\"): " + sb);

        sb.replace(0, 5, "Hi");
        System.out.println("After replace(0, 5, \"Hi\"): " + sb);

        sb.delete(2, 8);
        System.out.println("After delete(2, 8): " + sb);

        sb.reverse();
        System.out.println("After reverse(): " + sb);
        sb.reverse(); // put it back for the next step
        System.out.println("After reverse() again: " + sb);

        System.out.println("charAt(0): " + sb.charAt(0));
        System.out.println("length(): " + sb.length());

        // Only convert to an immutable String at the very end, once the building
        // is finished.
        String finalResult = sb.toString();
        System.out.println("Final String (via toString()): " + finalResult);

        // StringBuffer has the exact same API as StringBuilder, but every method
        // is synchronized -- it predates StringBuilder (Java 1.0 vs Java 5) and is
        // now mostly legacy: use it only if multiple threads truly share ONE
        // builder instance, otherwise StringBuilder is faster.
        StringBuffer buffer = new StringBuffer("legacy, thread-safe");
        buffer.append(" version");
        System.out.println("StringBuffer (synchronized, legacy): " + buffer);
    }
}
```

Expected Output

```
Initial: Hello
After append(): Hello, World
After append(char): Hello, World!
After insert(5, " there"): Hello there, World!
After replace(0, 5, "Hi"): Hi there, World!
After delete(2, 8): Hi, World!
After reverse(): !dlroW ,iH
After reverse() again: Hi, World!
charAt(0): H
length(): 10
Final String (via toString()): Hi, World!
StringBuffer (synchronized, legacy): legacy, thread-safe version
```

> `StringBuffer` has the exact same API as `StringBuilder`, but every method is synchronized (thread-safe) -- it has existed since Java 1.0, while `StringBuilder` (Java 5) does the same job without that synchronization OVERHEAD. `StringBuilder` should be preferred except in the rare case where multiple threads genuinely share the SAME builder instance.

Formatting: String.format() and Text Blocks

`String.format()` (or Java 15's `formatted()` instance method) provides printf-style formatting with placeholders like `%s` / `%d` / `%.2f`. Text blocks (`"""..."""`, Java 15+) let you define multi-line strings (like JSON, SQL, HTML) without the hassle of writing `\n` at the end of every line and `\"` around every quote.

```
public class StringFormattingExample {
    public static void main(String[] args) {
        String name = "Alice";
        int age = 30;
        double price = 19.999;

        // String.format() -- printf-style placeholders: %s (String), %d (int),
        // %.2f (double with 2 decimal places), etc.
        String formatted = String.format("%s is %d years old.", name, age);
        System.out.println(formatted);

        String priceLine = String.format("Price: $%.2f", price);
        System.out.println(priceLine);

        // Padding and alignment: %-10s left-aligns in a 10-char field, %10s
        // right-aligns.
        System.out.println(String.format("[%-10s][%10s]", "left", "right"));

        // formatted() (Java 15+) is the same thing as an instance method on the
        // format String itself -- purely a matter of style.
        String sameThing = "%s is %d years old.".formatted(name, age);
        System.out.println("Same result via formatted(): " + sameThing.equals(formatted));

        // Text blocks (Java 15+): a multi-line String literal delimited by """,
        // useful for embedded JSON/SQL/HTML without escaping every quote and
        // newline.
        String json = """
                {
                  "name": "%s",
                  "age": %d
                }""".formatted(name, age);
        System.out.println("Text block (formatted JSON):");
        System.out.println(json);

        // Text blocks strip "incidental" leading whitespace based on the least
        // indented line, and a trailing "\"\"\"" on its own line adds a final
        // newline -- both demonstrated by comparing lengths here.
        String withTrailingNewline = """
                line one
                line two
                """;
        String withoutTrailingNewline = """
                line one
                line two""";
        System.out.println("With trailing newline, length: " + withTrailingNewline.
length());
        System.out.println("Without trailing newline, length: " + withoutTrailingNewline.
length());
    }
}
```

> `String.format("%.2f", ...)` ROUNDS the decimal number, it doesn't truncate it -- in the example above, `19.999` with `%.2f` becomes `"20.00"` (standard rounding, not truncation). Also, in text blocks, whether the closing `"""` is on its own line or right after the last line of text affects whether a trailing newline (`\n`) is included -- in the example above, the two strings' lengths differ because of this (18 vs. 17).

Searching, Splitting, and Other Helper Methods

`split()` breaks a string into pieces based on a regex delimiter, and `String.join()` does the opposite (joins pieces together with a delimiter). `replace()` replaces ALL occurrences of a substring (literal, not regex); `replaceAll()` / `replaceFirst()` use regex instead. `trim()` only strips ASCII whitespace, while `strip()` (Java 11+) is Unicode-aware and is now generally preferred.

```
import java.util.Arrays;
import java.util.List;

public class StringSearchSplitExample {
    public static void main(String[] args) {
        String csv = "apple,banana,,cherry";

        // split() cuts a String into an array using a regex delimiter -- an empty
        // field (two delimiters in a row) still produces an empty String element.
        String[] parts = csv.split(",");
        System.out.println("split(\",\"): " + Arrays.toString(parts));
        System.out.println("parts.length: " + parts.length);

        // String.join() is the reverse operation: glue elements back together
        // with a separator.
        String joined = String.join(" | ", parts);
        System.out.println("String.join(\" | \", parts): " + joined);

        List<String> fromList = List.of("x", "y", "z");
        System.out.println("String.join(\"-\", List): " + String.join("-", fromList));

        // replace() replaces ALL occurrences of a literal substring (no regex).
        String sentence = "the cat sat on the mat";
        System.out.println("replace(\"at\", \"og\"): " + sentence.replace("at", "og"));

        // replaceAll() takes a REGEX, replaceFirst() only replaces the first match.
        System.out.println("replaceAll(\"\\\\s+\", \"_\"): " + sentence.replaceAll("\\s+",
"_"));

        // trim() removes ASCII whitespace (<= U+0020) from both ends; strip()
        // (Java 11+) is the Unicode-aware version and is generally preferred now.
        String padded = "   spaced out   ";
        System.out.println("trim(): [" + padded.trim() + "]");
        System.out.println("strip(): [" + padded.strip() + "]");

        // Case conversion and equality that ignores case.
        System.out.println("\"Hello\".equalsIgnoreCase(\"HELLO\"): " + "Hello".
equalsIgnoreCase("HELLO"));
        System.out.println("toLowerCase(): " + "MixedCase".toLowerCase());

        // repeat() (Java 11+) builds a String by repeating it N times.
        System.out.println("\"ab\".repeat(3): " + "ab".repeat(3));

        // compareTo() is lexicographic (dictionary) order -- negative, zero, or
        // positive, like Comparable in general.
        System.out.println("\"apple\".compareTo(\"banana\"): " + "apple".compareTo
("banana"));
        System.out.println("\"banana\".compareTo(\"apple\"): " + "banana".compareTo
("apple"));
        System.out.println("\"apple\".compareTo(\"apple\"): " + "apple".compareTo("apple"));
    }
}
```

Expected Output

```
split(","): [apple, banana, , cherry]
parts.length: 4
String.join(" | ", parts): apple | banana |  | cherry
String.join("-", List): x-y-z
replace("at", "og"): the cog sog on the mog
replaceAll("\\s+", "_"): the_cat_sat_on_the_mat
trim(): [spaced out]
strip(): [spaced out]
"Hello".equalsIgnoreCase("HELLO"): true
toLowerCase(): mixedcase
"ab".repeat(3): abababab
"apple".compareTo("banana"): -1
"banana".compareTo("apple"): 1
"apple".compareTo("apple"): 0
```

Best Practices

- **Always use `equals()` / `equalsIgnoreCase()` when comparing string content, never `==`** -- `==` only compares object identity, and while it may happen to give the right answer in some cases, it isn't reliable.
- **Use `StringBuilder` if you're concatenating many strings in a loop**, not `+` / `+=` -- `+` is fine for one-off, short concatenations (which the compiler can optimize), but leads to O(n^2) cost inside a loop.
- **Don't hesitate to use `String` as a `HashMap` key** -- thanks to immutability and a cached `hashCode()`, this is a safe and fast thing to do.
- **Use a text block (`"""`) for long, multi-line text (JSON, SQL, HTML)** -- it's far more readable than a single-line string full of escape characters (`\"`, `\n`).

Common Mistakes

- **Comparing string content with `==` and sometimes getting the right answer, sometimes not.** Two string literals being `==` equal is a coincidence (thanks to the string pool), not a general rule -- `equals()` should always be used.
- **Forgetting to assign a `String` method's return value to a variable.** `str.trim();` on its own does nothing -- you need `str = str.trim();`, because `String` is immutable.
- **Accumulating a string with `+` / `+=` inside a loop.** Unnoticeable at a small scale, but leads to O(n^2) cost in large loops -- `StringBuilder` should be used instead.
- **Assuming `String.format("%.2f", ...)` truncates.** It actually performs standard rounding (`19.999` → `"20.00"`), which can sometimes lead to unexpected results.

Summary, Cheat Sheet, and Glossary

`String` is an IMMUTABLE class in Java that represents a sequence of characters -- every "modifying" method actually returns a new `String`. The JVM shares string literals through a

"string pool"; because of this, `==` sometimes (misleadingly) gives the right answer, but `equals()` should always be used for content comparison. `StringBuilder` should be preferred when frequent string concatenation is needed; `String.format()` / `formatted()` can be used for formatting, and text blocks for multi-line text.

Quick reference:

```
String a = "hello";                          // literal -- shared via the string pool
String b = new String("hello");               // always a new, separate object
a.equals(b);                                    // CONTENT comparison -- true
a == b;                                           // IDENTITY comparison -- false

StringBuilder sb = new StringBuilder();             // mutable, for concatenation in a loop
sb.append("x").append("y");

String.format("%s: %.2f", "total", 19.999);            // "total: 20.00" (rounded)

String multi = """
        multi-line
        text""";                                          // text block (Java 15+)
```

Glossary

String — An immutable class in Java that represents a sequence of characters.

Immutable — A design where an object's content never changes once created; every "modifying" operation returns a new object instead.

String Pool — A special memory area (intern table) where the JVM stores and shares string literals with the same text.

StringBuilder — `String`'s mutable sibling, used for building a string piece by piece.

Text Block — Multi-line string literal syntax delimited by `"""`, introduced in Java 15.