

UNIT 6

Handling Strings

(2 Hours) — Object Oriented Programming in Java (CACS204)

Text is one of the most common kinds of data a program handles, and Java represents it using the String class. This unit covers how Strings are created, concatenated, and converted; how to change case, extract characters, compare, search, and modify String content; and the mutable alternative, StringBuffer, used when a program needs to build or change text efficiently.

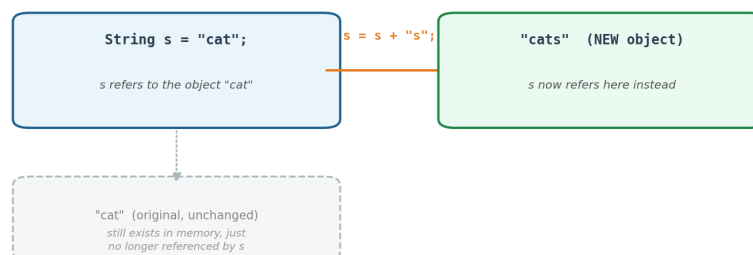
6.1 Creation, Concatenation and Conversion of a String

6.1.1 Creating Strings

A String in Java is an object representing an immutable sequence of characters. It can be created directly from a string literal, or explicitly using the new keyword.

```
String s1 = "hello"; // string literal — may be shared from a pool
String s2 = new String("hello"); // explicit object — always a distinct object
char[] letters = {'J', 'a', 'v', 'a'};
String s3 = new String(letters); // built from a char array -> "Java"
```

Note: Java maintains a string pool for literals: identical string literals used anywhere in a program may refer to the exact same object in memory, unlike Strings created with new, which always produce a fresh object.



Strings in Java are IMMUTABLE — no method ever changes a String's characters; operations always produce a new String object

Figure 6.1 — String objects are immutable: any operation that appears to change one actually produces a new String, leaving the original untouched.

6.1.2 Concatenation

Concatenation joins two or more Strings together. It can be performed with the + operator, or with the concat() method. Because Strings are immutable, every concatenation produces a brand-new String object.

```
String first = "Object";
String second = "Oriented";
String combined = first + " " + second; // "Object Oriented"
String combined2 = first.concat(" ").concat(second); // same result
```

The + operator also converts non-String operands automatically when at least one operand is a String:

```
int age = 20;
String msg = "Age: " + age; // "Age: 20" (int converted to String automatically)
```

6.1.3 Conversion

Values are commonly converted to a String using concatenation or String.valueOf(), and from a String back to a primitive type using wrapper class methods such as Integer.parseInt() or Double.parseDouble().

Direction	Common method(s)	Example
Number -> String	String.valueOf(x), x + ""	String.valueOf(42) -> "42"
String -> int	Integer.parseInt(s)	Integer.parseInt("42") -> 42
String -> double	Double.parseDouble(s)	Double.parseDouble("3.14") -> 3.14
char[] -> String	new String(charArray)	new String({'H','i'}) -> "Hi"
String -> char[]	s.toCharArray()	"Hi".toCharArray() -> ['H','i']

Note: Converting an invalid numeric String, such as Integer.parseInt("abc"), throws a NumberFormatException (Unit 5) — this should be anticipated with a try/catch block when parsing untrusted input.

6.2 Changing Case

The methods toUpperCase() and toLowerCase() return a new String with every letter converted to the requested case, leaving non-letter characters unchanged. As with all String methods, the original String is never modified.

```
String name = "Java Programming";
System.out.println(name.toUpperCase()); // "JAVA PROGRAMMING"
System.out.println(name.toLowerCase()); // "java programming"
System.out.println(name); // still "Java Programming" — unchanged
```

Example 6.1

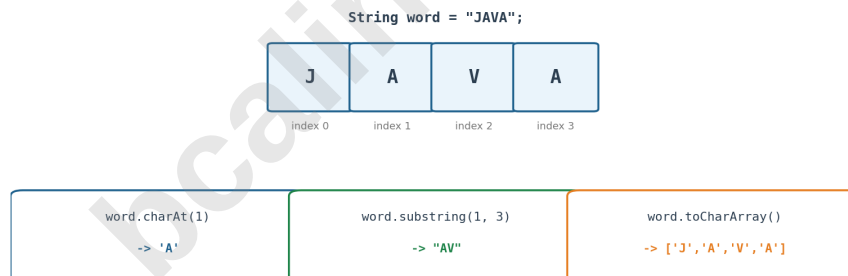
Write an expression that checks whether two Strings are equal, ignoring case differences.

Solution

```
String a = "Hello";
String b = "HELLO";
boolean same = a.equalsIgnoreCase(b); // true — a dedicated case-insensitive comparison
```

6.3 Character Extraction

Individual characters or substrings can be pulled out of a String using several methods, all of which treat the String as zero-indexed — the first character sits at index 0.



Character positions are zero-indexed: the first character is at index 0

Figure 6.2 — Extracting characters and substrings from "JAVA" by index.

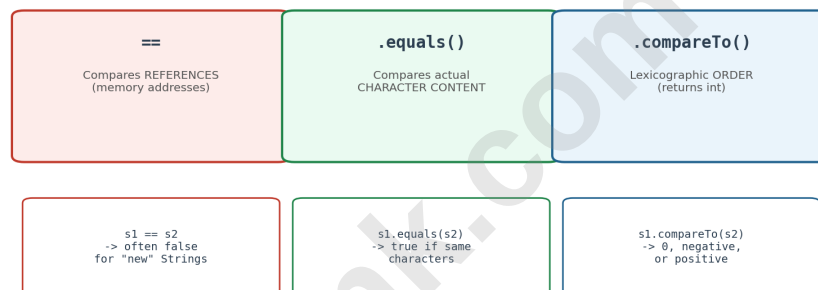
Method	Returns
charAt(int index)	The single char at the given index
substring(int begin)	Everything from begin to the end of the String
substring(int begin, int end)	Characters from begin (inclusive) to end (exclusive)
toCharArray()	The entire String as a char[] array
length()	The number of characters in the String

```
String word = "Programming";
char c = word.charAt(0); // 'P'
String sub = word.substring(0, 4); // "Prog"
int len = word.length(); // 11
```

Note: substring(begin, end) does NOT include the character at index end — this is a common off-by-one mistake to watch for.

6.4 String Comparison

Java provides several distinct ways to compare Strings, and choosing the wrong one is one of the most common sources of bugs for beginners — particularly confusing `==` with `.equals()`.



Rule of thumb: always use `.equals()` to compare String content, never `==`

Figure 6.3 — `==` compares object references; `.equals()` compares character content; `.compareTo()` gives lexicographic ordering.

```
String s1 = new String("cat");
String s2 = new String("cat");
System.out.println(s1 == s2); // false — two different objects
System.out.println(s1.equals(s2)); // true — same characters
System.out.println(s1.compareTo(s2)); // 0 — equal, lexicographically
System.out.println("cat".compareTo("dog")); // negative — "cat" precedes "dog"
```

Comparison Method Summary

- ✓ `==` compares references (memory identity) — unreliable for String content
- ✓ `.equals(Object)` compares character content — always use this for equality checks
- ✓ `.equalsIgnoreCase(String)` — content equality, ignoring letter case
- ✓ `.compareTo(String)` — returns negative/zero/positive for lexicographic ordering

6.5 Searching Strings

Several methods locate characters or substrings within a larger String, either returning a position (an index) or a simple true/false result.

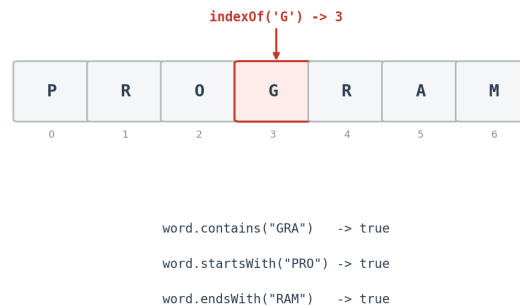


Figure 6.4 — Locating a character and checking for substrings within "PROGRAM".

Method	Purpose
indexOf(String/char)	Index of the first occurrence, or -1 if not found
lastIndexOf(String/char)	Index of the last occurrence, or -1 if not found
contains(CharSequence)	true if the given substring appears anywhere within
startsWith(String)	true if the String begins with the given prefix
endsWith(String)	true if the String ends with the given suffix

```
String text = "the quick brown fox";  
int pos = text.indexOf("brown"); // 10  
boolean has = text.contains("fox"); // true  
boolean starts = text.startsWith("the"); // true
```

Example 6.2

Given String path = "report_final.pdf", write an expression that extracts just the file extension ("pdf") using indexOf() and substring().

Solution

```
String path = "report_final.pdf";  
int dot = path.lastIndexOf('.');  
String extension = path.substring(dot + 1); // "pdf"
```

6.6 Modifying Strings

Since Strings are immutable, no method truly "modifies" one in place — instead, each of the following methods returns a new String reflecting the requested change, which is typically reassigned back to the same variable.

Method	Effect
<code>trim()</code>	Removes leading and trailing whitespace
<code>replace(old, new)</code>	Replaces all occurrences of a character or substring
<code>concat(String)</code>	Appends another String, returning the combined result
<code>toUpperCase()</code> / <code>toLowerCase()</code>	Returns the String with case changed
<code>split(String regex)</code>	Breaks the String into an array of substrings

```
String raw = " Hello, World! ";
String trimmed = raw.trim(); // "Hello, World!"
String replaced = trimmed.replace('!', '?'); // "Hello, World?"
String[] parts = trimmed.split(", "); // ["Hello", "World!"]
```

Note: Because each of these methods returns a new String rather than altering the original, a common mistake is writing `str.trim();` alone and expecting `str` itself to change — the result must be captured, e.g. `str = str.trim();`

6.7 String Buffer

Repeatedly concatenating Strings inside a loop is inefficient, since every concatenation silently creates a new object and discards the old one. `StringBuffer` (and its newer, non-thread-safe relative `StringBuilder`) solves this by representing a mutable sequence of characters that can be changed in place, without creating a new object for every edit.

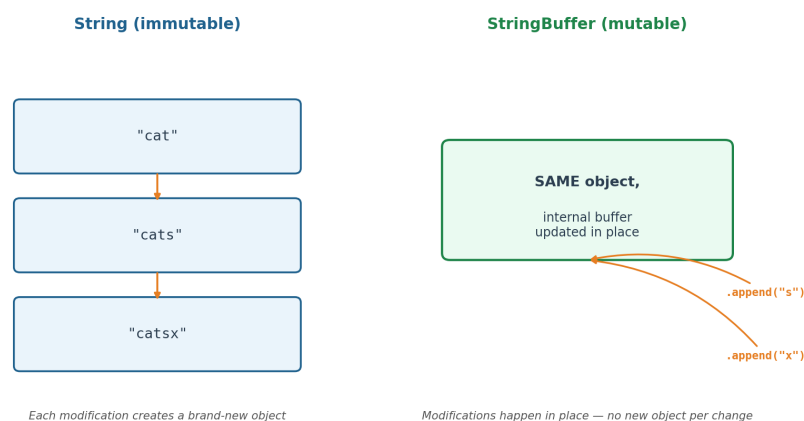


Figure 6.5 — `String` creates a new object per change; `StringBuffer` edits the same object's internal buffer directly.

Method	Effect
append(x)	Adds x to the end of the buffer
insert(index, x)	Inserts x at the given position
delete(start, end)	Removes characters from start (inclusive) to end (exclusive)
reverse()	Reverses the character sequence in place
replace(start, end, str)	Replaces a range of characters with str
toString()	Converts the buffer's contents back into an immutable String

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(", World!"); // sb now holds "Hello, World!" – SAME object
sb.insert(0, ">> "); // ">> Hello, World!"
sb.reverse(); // "!dlroW ,olleH >>"
System.out.println(sb.length()); // buffer length, updates as content
changes
String finalStr = sb.toString(); // convert back to an immutable String
when done
```

Example 6.3

Use a `StringBuffer` to build the String "1-2-3-4-5" by appending numbers and hyphens in a loop, rather than concatenating Strings directly.

Solution

```
StringBuffer sb = new StringBuffer();
for (int i = 1; i <= 5; i++) {
    sb.append(i);
    if (i < 5) sb.append("-");
}
String result = sb.toString(); // "1-2-3-4-5"
```

Note: Prefer `StringBuffer` or `StringBuilder` over repeated + concatenation whenever building up a String incrementally, especially inside loops — it avoids creating and discarding many intermediate String objects.

Key Takeaways

- ✓ Strings are immutable — every operation returns a new String rather than modifying the original
- ✓ + and concat() join Strings; String.valueOf()/parseInt()/parseDouble() convert between types
- ✓ toUpperCase()/toLowerCase() change case without altering the original String
- ✓ charAt(), substring(), and toCharArray() extract characters using zero-based indexing
- ✓ Use .equals() (never ==) to compare String content; compareTo() gives lexicographic order
- ✓ indexOf(), contains(), startsWith(), and endsWith() search within a String
- ✓ trim(), replace(), and split() return modified copies, which must be reassigned to keep the change
- ✓ StringBuffer offers a mutable, in-place alternative — far more efficient for repeated edits

— *End of Unit 6* —