Chapter 3.4: Strings

Object Oriented Programming with Java

The String object · Indexing · Basic methods · Immutability · StringBuilder · Input with Scanner · == and equals() · String pool · compareTo · printf and formatting

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_03_04_Strings.html

INTRODUCTION

Learning Outcomes

Creating strings

`String` objects: create them from literals and with `new`; explain the order of `+` concatenation.

Indexing and methods

`charAt`, `length`, `indexOf`, `substring`: access parts of a string and loop over its characters.

Immutability

Show in memory that methods return new objects; use `StringBuilder` in loops.

User input

`next()`, `nextLine()`, `nextInt()`: explain how they differ and avoid the leftover-newline trap.

Comparison

`==` vs. `equals()`: explain the difference using the string pool; use `compareTo` and related methods.

Formatted output

`printf` and `String.format`: set width, alignment and precision.

STRING BASICS

What Is a String?

- **A string** is an **object** that holds a sequence of characters; its type is the class `java.lang.String`.
- `String` is a **class**, not a primitive type; its name is capitalized. `import` is not needed.
- Text in double quotes, like `"Tarkan"`, is a **string literal**; the object is created without `new`.
- Characters are stored as **Unicode**, so Turkish letters such as `ç ğ ı İ ö ş ü` work fine.
- A single character is a `char` in single quotes (`'T'`); text is a `String` in double quotes (`"T"`).

```
String name = "Tarkan";
String fullName = "Mustafa Sandal";
String point = "(" + 3 + ", " + 5 + ")";
System.out.println(name);
System.out.println(fullName);
System.out.println(point);
```

OUTPUT

```
Tarkan
Mustafa Sandal
(3, 5)
```



A string is like a row of character boxes.

Creating and Concatenating Strings (+)

```
String s0 = "Giresun";            // literal
char[] letters = {'J', 'a', 'v', 'a'};
String s1 = new String(letters);  // char[]
String s2 = String.valueOf(42);   // from int
String s3 = "Giresun".concat(" Uni.");
System.out.println(s1 + " " + s2 + " " + s3);

System.out.println("Sum: " + 3 + 5);
System.out.println("Sum: " + (3 + 5));
System.out.println(3 + 5 + " apples");
System.out.println('a' + 'b' + "c");
System.out.println("" + 'a' + 'b');
```

OUTPUT

```
Java 42 Giresun Uni.
Sum: 35
Sum: 8
8 apples
195c
ab
```

- `+` : if either operand is a `String`, the other is converted to a string and the two are **concatenated**.
- `+` **is evaluated left to right**: `"Sum: " + 3` first becomes `"Sum: 3"`, then `5` is appended → `Sum: 35`.
- Parentheses give the sum: `"Sum: " + (3 + 5)`.
- `3 + 5 + " apples"` : the numbers are added first (8), then the string is appended.
- `'a' + 'b'` adds two `char` values as **numbers** (97 + 98 = 195). Putting `""` first forces string concatenation.
- Every concatenation creates a **new** `String` object.

+ Left to Right: Addition or Concatenation?

```
▶ int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```

a = 4 b = 2

int int

Two `int` variables. On each line, `+` **is evaluated left to right**; the type is checked at every step.

Step 1 / 13

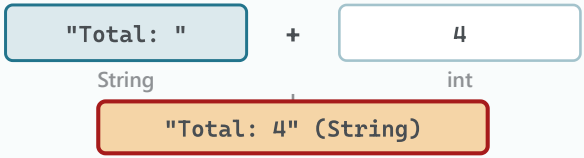
VARIABLES

`a = 4` `b = 2`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
▶ String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



Left side is a String: 4 is converted and appended → `"Total: 4"`.

Step 2 / 13

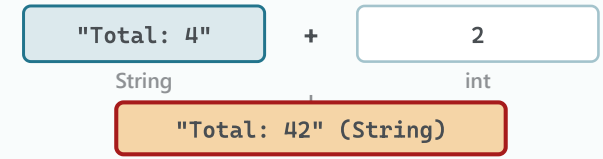
VARIABLES

`a = 4` `b = 2`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
▶ String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



Still a String; 2 is appended as text too → `"Total: 42"`, not a sum!

Step 3 / 13

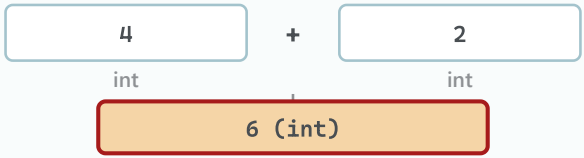
VARIABLES

`s1 = Total: 42`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
▶ String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



Parentheses first: `4 + 2` adds two ints: the numeric sum is 6.

Step 4 / 13

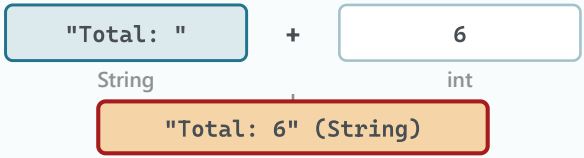
VARIABLES

`s1 = Total: 42`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
▶ String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



Then concatenated with the string → `"Total: 6"`.

Step 5 / 13

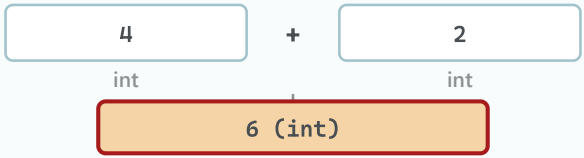
VARIABLES

`s1 = Total: 42` `s2 = Total: 6`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
▶ String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



Both left operands are int: added first → 6.

Step 6 / 13

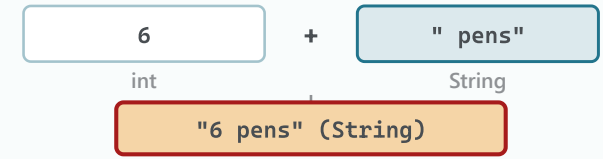
VARIABLES

`s1 = Total: 42` `s2 = Total: 6`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
▶ String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



6 + `" pens"` : now a String → `"6 pens"` .

Step 7 / 13

VARIABLES

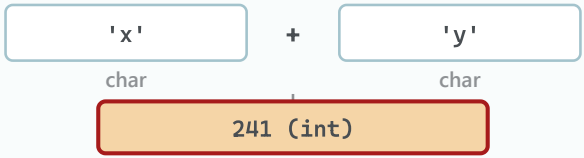
`s1 = Total: 42` `s2 = Total: 6`

`s3 = 6 pens`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
▶ String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



`'x' + 'y'` : two chars are added as numbers: 120 + 121 = 241.

Step 8 / 13

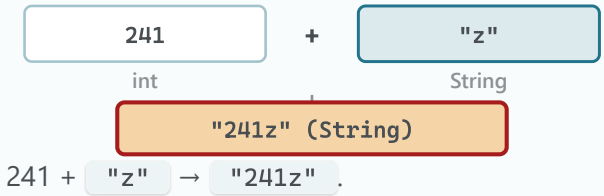
VARIABLES

`s1` = **Total: 42** `s2` = **Total: 6** `s3` = **6 pens**

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
▶ String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



241 + `"z"` → `"241z"` .

Step 9 / 13

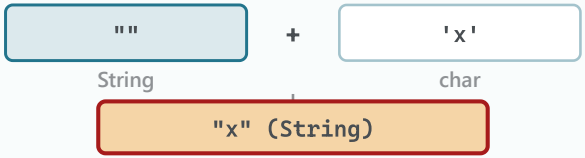
VARIABLES

`s1` = **Total: 42** `s2` = **Total: 6** `s3` = **6 pens** `s4` = **241z**

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
▶ String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



Empty string first: `"" + 'x'` becomes a String → `"x"`.

Step 10 / 13

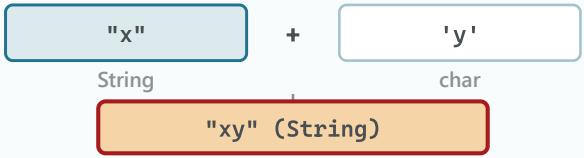
VARIABLES

`s1` = **Total: 42** `s2` = **Total: 6** `s3` = **6 pens** `s4` = **241z**

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
▶ String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```



`"x" + 'y'` → `"xy"` : the char is appended as text.

Step 11 / 13

VARIABLES

`s1 = Total: 42` `s2 = Total: 6` `s3 = 6 pens` `s4 = 241z` `s5 = xy`

OUTPUT

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
▶ System.out.println(s1 + " | " + s2);
8 System.out.println(s3 + " | " + s4 + " | " + s5);
```

println

concatenated strings are printed

The first line is printed.

Step 12 / 13

VARIABLES

s1 = **Total: 42** | s2 = **Total: 6** | s3 = **6 pens** | s4 = **241z** | s5 = **xy**

OUTPUT

```
Total: 42 | Total: 6
```

+ Left to Right: Addition or Concatenation?

```
1 int a = 4, b = 2;
2 String s1 = "Total: " + a + b;
3 String s2 = "Total: " + (a + b);
4 String s3 = a + b + " pens";
5 String s4 = 'x' + 'y' + "z";
6 String s5 = "" + 'x' + 'y';
7 System.out.println(s1 + " | " + s2);
▶ System.out.println(s3 + " | " + s4 + " | " + s5);
```

println

concatenated strings are printed

The second line is printed. Rule: once one side is a String, + concatenates; until then, numbers are added.

Step 13 / 13

VARIABLES

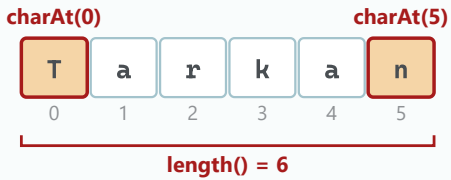
`s1 = Total: 42` `s2 = Total: 6` `s3 = 6 pens` `s4 = 241z` `s5 = xy`

OUTPUT

```
Total: 42 | Total: 6
6 pens | 241z | xy
```

Indexing

- The first character has index **0**; the last has index `length() - 1`.
- `charAt(i)`: character at index `i` (a `char`).
- `length()` is a **method** and needs parentheses (an array's `length` is a field).
- An index past the end or below zero throws a runtime `StringIndexOutOfBoundsException`.
- `char` is a number (its Unicode code): `c + 1` is a number, `(char) (c + 1)` is the next letter.




```
String name = "Tarkan";
System.out.println(name.length());
System.out.println(name.charAt(0));
System.out.println(name.charAt(name.length() - 1));
char c = name.charAt(2);
System.out.println(c + 1);
System.out.println((char) (c + 1));
```

OUTPUT

```
6
T
n
115
s
```

```
String s = "Java";
System.out.println(s.charAt(4));
```

```
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException:
    Index 4 out of bounds for length 4
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
▶ System.out.print("Enter a word: ");
3 String word = scanner.nextLine();
4 for (int i = 0; i < word.length(); i += 2) {
5   char letter = word.charAt(i);
6   System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
8 scanner.close();
```

word not read yet

The prompt is printed; `print` does not start a new line.

Step 1 / 8

VARIABLES

—

OUTPUT

```
Enter a word:
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a word: ");
▶ String word = scanner.nextLine();
4 for (int i = 0; i < word.length(); i += 2) {
5   char letter = word.charAt(i);
6   System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
8 scanner.close();
```



The user types **Giresun** and presses Enter; `nextLine()` reads the whole line.

Step 2 / 8

VARIABLES

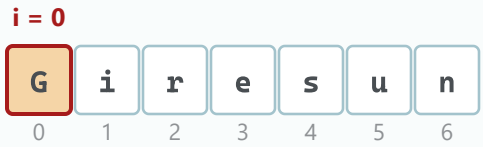
`word = "Giresun"`

OUTPUT

```
Enter a word: Giresun
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a word: ");
3 String word = scanner.nextLine();
4 for (int i = 0; i < word.length(); i += 2) {
    char letter = word.charAt(i);
    System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
8 scanner.close();
```



`i = 0` : `charAt(0)` = **'G'**; the line is printed, then `i += 2` .

Step 3 / 8

VARIABLES

`word = "Giresun"` `i = 0`

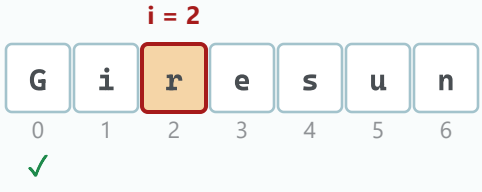
`letter = 'G'`

OUTPUT

```
Enter a word: Giresun
Letter 1: G
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a word: ");
3 String word = scanner.nextLine();
4 for (int i = 0; i < word.length(); i += 2) {
▶   char letter = word.charAt(i);
▶   System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
8 scanner.close();
```



✓

`i = 2` : `charAt(2)` = **'r'**; the line is printed, then `i += 2` .

Step 4 / 8

VARIABLES

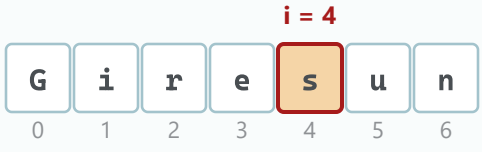
`word = "Giresun"` `i = 2` `letter = 'r'`

OUTPUT

```
Enter a word: Giresun
Letter 1: G
Letter 3: r
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a word: ");
3 String word = scanner.nextLine();
4 for (int i = 0; i < word.length(); i += 2) {
▶   char letter = word.charAt(i);
▶   System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
8 scanner.close();
```



`i = 4` : `charAt(4)` = **'s'**; the line is printed, then `i += 2` .

Step 5 / 8

VARIABLES

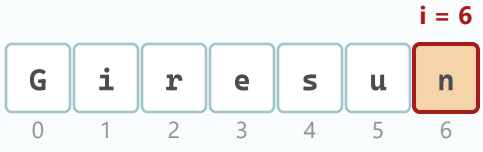
`word` = **"Giresun"** `i` = **4** `letter` = **'s'**

OUTPUT

```
Enter a word: Giresun
Letter 1: G
Letter 3: r
Letter 5: s
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a word: ");
3 String word = scanner.nextLine();
4 for (int i = 0; i < word.length(); i += 2) {
▶   char letter = word.charAt(i);
▶   System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
8 scanner.close();
```



`i = 6` : `charAt(6)` = **'n'**; the line is printed, then `i += 2` .

Step 6 / 8

VARIABLES

`word = "Giresun"` `i = 6` `letter = 'n'`

OUTPUT

```
Enter a word: Giresun
Letter 1: G
Letter 3: r
Letter 5: s
Letter 7: n
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a word: ");
3 String word = scanner.nextLine();
▶ for (int i = 0; i < word.length(); i += 2) {
5   char letter = word.charAt(i);
6   System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
8 scanner.close();
```

i = 8 → 8 < 7 false

G	i	r	e	s	u	n
0	1	2	3	4	5	6
✓		✓		✓		✓

`i = 8` : `8 < 7` is false: the loop ends. The last index is 6; 7 is never used.

Step 7 / 8

VARIABLES

`word = "Giresun"` `i = 8`

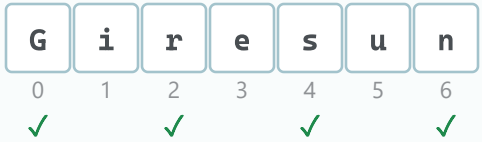
OUTPUT

```
Enter a word: Giresun
Letter 1: G
Letter 3: r
Letter 5: s
Letter 7: n
```

Looping over Characters: Step by Step

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a word: ");
3 String word = scanner.nextLine();
4 for (int i = 0; i < word.length(); i += 2) {
5   char letter = word.charAt(i);
6   System.out.println("Letter " + (i + 1) + ": " +
  letter);
7 }
▶ scanner.close();
```

i = 8 → 8 < 7 false



The Scanner is closed and the program ends.

Step 8 / 8

VARIABLES

`word = "Giresun"`

OUTPUT

```
Enter a word: Giresun
Letter 1: G
Letter 3: r
Letter 5: s
Letter 7: n
```

STRING METHODS

Basic String Methods

Method	What it does	For `s = "Java Programming"`
`length()`	Number of characters.	`16`
`charAt(i)`	Character at index `i` (a `char`).	`s.charAt(0)` → `'J'`
`indexOf(str)`	Index of the **first** match; **`-1`** if absent.	`s.indexOf("Programming")` → `5` `s.indexOf("C++")` → `-1`
`indexOf(str, from)`	Searches from index `from`.	`s.indexOf("a", 4)` → `10`
`lastIndexOf(str)`	Index of the last match.	`s.lastIndexOf('g')` → `15`
`substring(i1, i2)`	From `i1` to `i2` (**exclusive**).	`s.substring(5, 12)` → `"Program"`
`substring(i1)`	`i1` to the end.	`s.substring(5)` → `"Programming"`
`toLowerCase()`	A **new** lowercase string.	`"java programming"`
`toUpperCase()`	A **new** uppercase string.	`"JAVA PROGRAMMING"`

No method changes `s`; the result always comes back as a new value.

indexOf and substring: Step by Step



A 16-character string; the space is a character too (index 4).

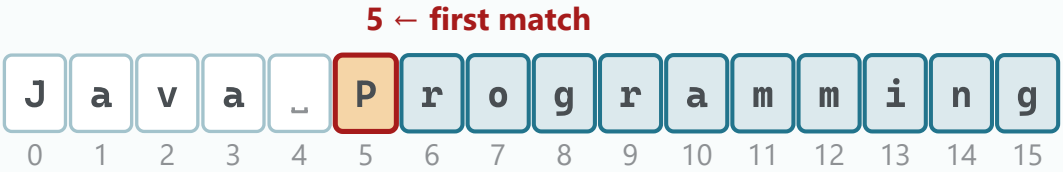
Step 1 / 7

```
▶ String str = "Java Programming";
2 int index = str.indexOf("Programming");
3 int len = str.length();
4 String sub = str.substring(5, 12);
5 String rest = str.substring(5);
6 String lower = str.toLowerCase();
7 String upper = str.toUpperCase();
```

VARIABLES

`str = "Java Programming"`

indexOf and substring: Step by Step



`indexOf("Programming")` returns the index where the match **starts**: 5.

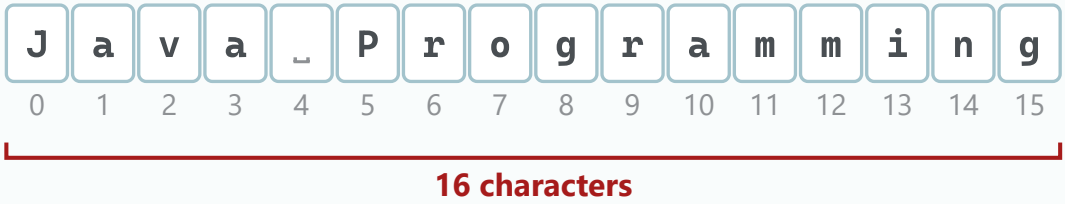
Step 2 / 7

```
1 String str = "Java Programming";
▶ int index = str.indexOf("Programming");
3 int len = str.length();
4 String sub = str.substring(5, 12);
5 String rest = str.substring(5);
6 String lower = str.toLowerCase();
7 String upper = str.toUpperCase();
```

VARIABLES

`str = "Java Programming"` `index = 5`

indexOf and substring: Step by Step



`length()` = 16; indices 0 … 15.

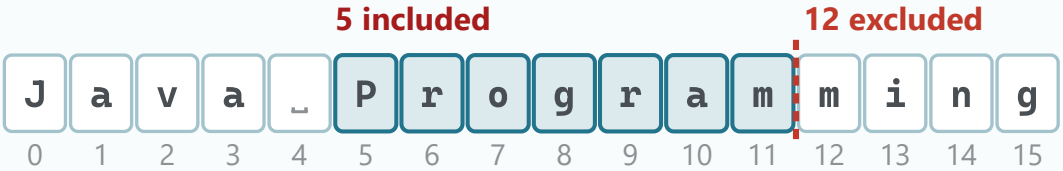
Step 3 / 7

```
1 String str = "Java Programming";
2 int index = str.indexOf("Programming");
▶ int len = str.length();
4 String sub = str.substring(5, 12);
5 String rest = str.substring(5);
6 String lower = str.toLowerCase();
7 String upper = str.toUpperCase();
```

VARIABLES

`str = "Java Programming"` `index = 5` `len = 16`

indexOf and substring: Step by Step



`substring(5, 12)` : 5, 6, …, 11 → **"Program"** (12 − 5 = 7 characters).

Step 4 / 7

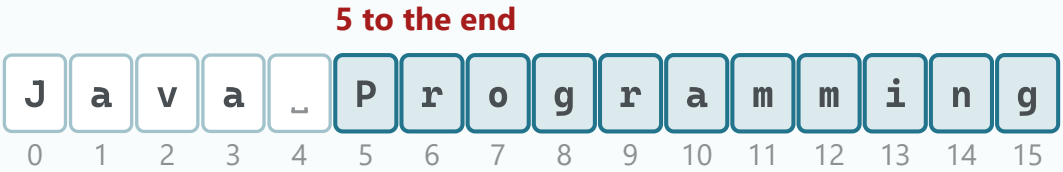
```
1 String str = "Java Programming";
2 int index = str.indexOf("Programming");
3 int len = str.length();
▶ String sub = str.substring(5, 12);
5 String rest = str.substring(5);
6 String lower = str.toLowerCase();
7 String upper = str.toUpperCase();
```

VARIABLES

`str = "Java Programming"` `index = 5`

`len = 16` `sub = "Program"`

indexOf and substring: Step by Step



`substring(5)` → **"Programming"**.

Step 5 / 7

```
1 String str = "Java Programming";
2 int index = str.indexOf("Programming");
3 int len = str.length();
4 String sub = str.substring(5, 12);
▶ String rest = str.substring(5);
6 String lower = str.toLowerCase();
7 String upper = str.toUpperCase();
```

VARIABLES

`str = "Java Programming"` `len = 16` `sub = "Program"` `rest = "Programming"`

indexOf and substring: Step by Step

new object



`toLowerCase()` **returns a new** string; `str` is still "Java Programming".

Step 6 / 7

```
1 String str = "Java Programming";
2 int index = str.indexOf("Programming");
3 int len = str.length();
4 String sub = str.substring(5, 12);
5 String rest = str.substring(5);
▶ String lower = str.toLowerCase();
7 String upper = str.toUpperCase();
```

VARIABLES

- `str = "Java Programming"`
- `sub = "Program"`
- `rest = "Programming"`
- `lower = "java programming"`

indexOf and substring: Step by Step



`toUpperCase()` also returns a new string. The original never changed.

Step 7 / 7

```
1 String str = "Java Programming";
2 int index = str.indexOf("Programming");
3 int len = str.length();
4 String sub = str.substring(5, 12);
5 String rest = str.substring(5);
6 String lower = str.toLowerCase();
▶ String upper = str.toUpperCase();
```

VARIABLES

`str = "Java Programming"`

`lower = "java programming"`

`upper = "JAVA PROGRAMMING"`

Other Useful Methods

```
String s = "  Hello World  ";
System.out.println("[" + s.trim() + "]");
System.out.println("[" + s.replace('o', '0') + "]");
String[] fruits = "apple,pear,plum".split(",");
System.out.println(fruits.length);
System.out.println(fruits[1]);
System.out.println(String.join(" - ", fruits));
System.out.println("ab".repeat(3));       // Java 11+
System.out.println("".isEmpty());
System.out.println("   ".isBlank());      // Java 11+
char[] arr = "Java".toCharArray();
System.out.println(arr.length + " " + arr[0]);
System.out.println("Giresun".replace("sun", "SUN"));
System.out.println(String.valueOf(3.5) + 1);
System.out.println(Integer.parseInt("42") + 1);
```

OUTPUT

```
[Hello World]
[  Hell0 W0rld  ]
3
pear
apple - pear - plum
ababab
true
true
4 J
GireSUN
3.51
43
```

- `trim()` : removes leading and trailing spaces. `replace` : replaces every match.
- `split(sep)` : returns the pieces as a `String[]` array; `String.join` does the reverse.
- `String.valueOf(x)` : number → string; `Integer.parseInt(s)` , `Double.parseDouble(s)` : string → number.

Immutable Objects

- `String` objects are **immutable**: they cannot change once created.
- `toUpperCase`, `replace`, `substring`, `+` do not change the original; they return **a new object**.
- To keep the result you must **assign** it: `s = s.toUpperCase();`
- Benefits: safe, consistent code (code sharing a string cannot break it), memory savings in the **string pool**, and safe sharing between threads.

Analogy: a string is a **stone tablet**. It cannot be scraped and fixed; a new tablet is carved and replaces the old one.




```
String s = "Giresun Üniversitesi";
s.toUpperCase();      // result is lost
System.out.println(s);
s = s.toUpperCase();  // stored in s
System.out.println(s);
```

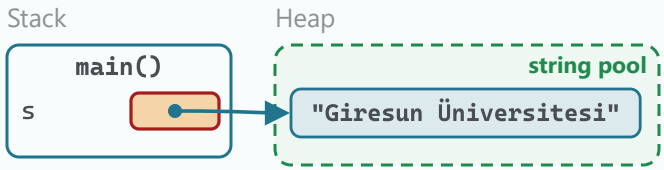
OUTPUT

```
Giresun Üniversitesi
GİRESUN ÜNİVERSİTESİ
```

On a Turkish-locale (tr-TR) machine the output is `GİRESUN ÜNİVERSİTESİ`; under an English locale it is `GIRESUN ÜNIVERSITESI`.

Immutability in Memory, Step by Step

```
▶ String s = "Giresun Üniversitesi";
2 s.toUpperCase();
3 System.out.println(s);
4 s = s.toUpperCase();
5 System.out.println(s);
```



The literal is an object in the pool; `s` points to it.

Step 1 / 5

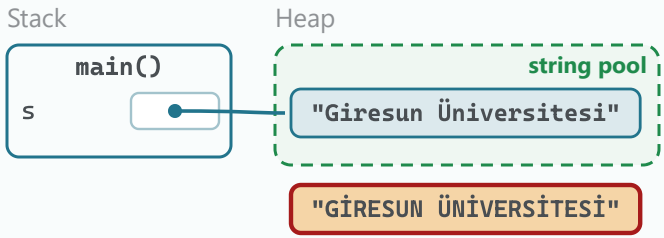
VARIABLES

`s = →#1 "Giresun Üniversitesi"`

OUTPUT

Immutability in Memory, Step by Step

```
1 String s = "Giresun Üniversitesi";
▶ s.toUpperCase();
3 System.out.println(s);
4 s = s.toUpperCase();
5 System.out.println(s);
```



`toUpperCase()` **creates a new** object. But the result is not assigned to any variable!

Step 2 / 5

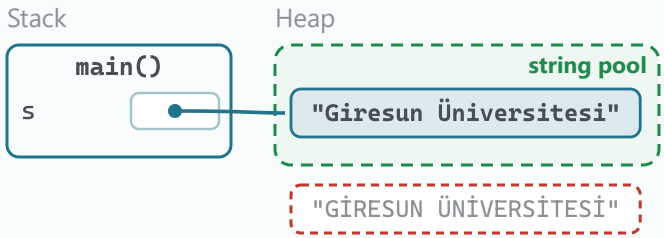
VARIABLES

`s = →#1 "Giresun Üniversitesi"`

OUTPUT

Immutability in Memory, Step by Step

```
1 String s = "Giresun Üniversitesi";
2 s.toUpperCase();
▶ System.out.println(s);
4 s = s.toUpperCase();
5 System.out.println(s);
```



`s` still points to the old object, so the original text is printed. The orphaned object is left to the garbage collector.

Step 3 / 5

VARIABLES

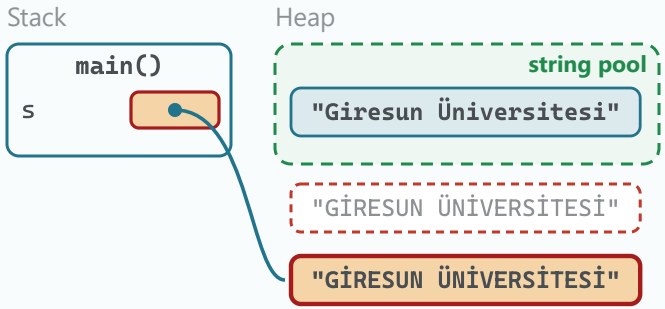
`s = →#1 "Giresun Üniversitesi"`

OUTPUT

```
Giresun Üniversitesi
```

Immutability in Memory, Step by Step

```
1 String s = "Giresun Üniversitesi";
2 s.toUpperCase();
3 System.out.println(s);
▶ s = s.toUpperCase();
5 System.out.println(s);
```



Again a new object is created, and this time **its reference is assigned to** `s`. The old object did not change; only the arrow moved.

Step 4 / 5

VARIABLES

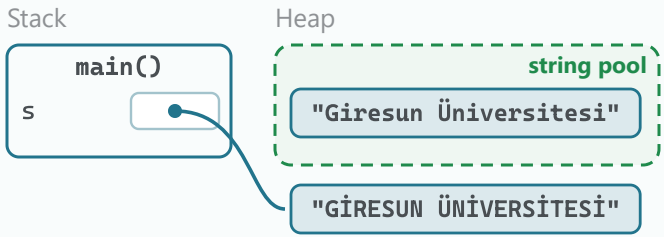
`s = →#3 "GİRESUN ÜNİVERSİTESİ"`

OUTPUT

```
Giresun Üniversitesi
```

Immutability in Memory, Step by Step

```
1 String s = "Giresun Üniversitesi";
2 s.toUpperCase();
3 System.out.println(s);
4 s = s.toUpperCase();
▶ System.out.println(s);
```



`s` now points to the uppercase object.

Step 5 / 5

VARIABLES

`s = →#3 "GİRESUN ÜNİVERSİTESİ"`

OUTPUT

```
Giresun Üniversitesi
GİRESUN ÜNİVERSİTESİ
```

Turkish Letters and the Locale

```java
import java.util.Locale;

String s = "Giresun Üniversitesi";
System.out.println(s.toUpperCase());
System.out.println(s.toUpperCase(Locale.ROOT));
System.out.println("TITLE".toLowerCase());
System.out.println("TITLE".toLowerCase().equals("title"));
System.out.println("TITLE".toLowerCase(Locale.ROOT));
```

OUTPUT (TR-TR)

```
GİRESUN ÜNİVERSİTESİ
GIRESUN ÜNIVERSITESI
tıtle
false
title
```

Conversion	Turkish (tr)	ROOT / English
`i` → upper	`İ`	`I`
`I` → lower	`ı`	`i`
`ı` → upper	`I`	`I`

- Without an argument, `toUpperCase()` / `toLowerCase()` use the machine's **default locale**: the same program can give different results on another machine.
- Under a Turkish locale, `"TITLE".toLowerCase()` → `"tıtle"`: the comparison is `false`!
- For commands, file names and similar text, pass `Locale.ROOT`; for Turkish text, `Locale.forLanguageTag("tr-TR")`.

Concatenation in Loops and StringBuilder

```
String result = "";
for (int i = 1; i <= 5; i++) {
  result += i + " ";   // new String each time
}
System.out.println(result);

StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++) {
  sb.append(i).append(' ');   // same object grows
}
String text = sb.toString();
System.out.println(text);

StringBuilder rev = new StringBuilder("Giresun");
rev.reverse();
System.out.println(rev);
StringBuilder k = new StringBuilder("Java");
k.insert(0, ">> ").append('!');
k.setCharAt(3, 'j');
System.out.println(k + " " + k.length());
```

OUTPUT

```
1 2 3 4 5
1 2 3 4 5
nuseriG
>> java! 8
```

- `result += ...` copies the old text on every iteration into a **new** object; n iterations copy about $n^2/2$ characters.
- `StringBuilder` is a **mutable** character buffer: `append`, `insert`, `reverse`, `setCharAt`, `deleteCharAt` all work on the same object.
- When done, `toString()` turns it into a `String`.
- For a few pieces, `+` is fine and readable; for many concatenations **inside a loop**, prefer `StringBuilder`.

StringBuilder: One Buffer, Changed in Place

```
▶ StringBuilder sb = new StringBuilder("Java");
2 sb.append(' ').append(8);
3 sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
7 String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



One mutable buffer: 4 characters used, capacity 16 + 4 = 20.

Step 1 / 9

VARIABLES

`length = 4` `capacity = 20`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
▶ sb.append(' ').append(8);
3 sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
7 String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



`append(' ')` : a space is appended; **same object**, no new object.

Step 2 / 9

VARIABLES

`length = 5` `capacity = 20`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
▶ sb.append(' ').append(8);
3 sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
7 String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



The second call in the chain, `append(8)`: the number is appended as a character. append returns the object itself, so calls can be chained.

Step 3 / 9

VARIABLES

`length = 6` `capacity = 20`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
2 sb.append(' ').append(8);
▶ sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
7 String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



`insert(0, ">> ")` : the other characters shift right to make room for 3 cells at the front.

Step 4 / 9

VARIABLES

`length = 9` `capacity = 20`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
2 sb.append(' ').append(8);
3 sb.insert(0, ">> ");
▶ sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
7 String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



`setCharAt(3, 'j')` : index 3 changes in place, which a String cannot do.

Step 5 / 9

VARIABLES

`length = 9` `capacity = 20`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
2 sb.append(' ').append(8);
3 sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
▶ sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
7 String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



`deleteCharAt(8)` : the trailing '8' is deleted; length 8.

Step 6 / 9

VARIABLES

`length = 8` `capacity = 20`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
2 sb.append(' ').append(8);
3 sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
▶ sb.append("programming book");
7 String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



16 more characters: 24 > 20, no room. The buffer grows: new capacity (20 + 1) × 2 = 42, old contents copied.

Step 7 / 9

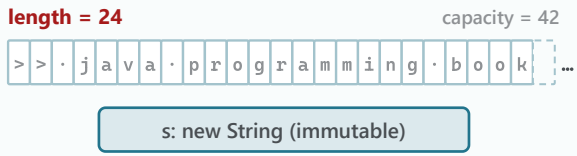
VARIABLES

`length = 24` `capacity = 42`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
2 sb.append(' ').append(8);
3 sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
▶ String s = sb.toString();
8 System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



`toString()` : creates **a new String** from the buffer's current contents.

Step 8 / 9

VARIABLES

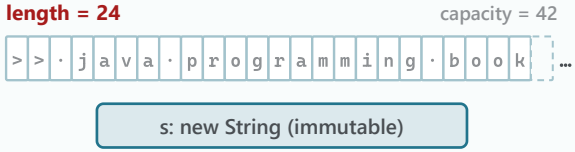
`length = 24` `capacity = 42`

`s = ">> java programming book"`

OUTPUT

StringBuilder: One Buffer, Changed in Place

```
1 StringBuilder sb = new StringBuilder("Java");
2 sb.append(' ').append(8);
3 sb.insert(0, ">> ");
4 sb.setCharAt(3, 'j');
5 sb.deleteCharAt(sb.length() - 1);
6 sb.append("programming book");
7 String s = sb.toString();
▶ System.out.println(s + " " + sb.length() + " " +
  sb.capacity());
```



Length 24 and capacity 42 are printed.

Step 9 / 9

VARIABLES

`length = 24` `capacity = 42`

`s = ">> java programming book"`

OUTPUT

```
>> java programming book 24 42
```

User Input: Scanner

- `java.util.Scanner` reads input from the keyboard (`System.in`): `import java.util.Scanner;`
- `next()` : skips whitespace and reads the next **word** (up to a space).
- `nextLine()` : reads **everything up to the end of the line**; it consumes the newline but drops it from the result.
- `nextInt()` , `nextDouble()` : reads a number but leaves the newline after it **in the buffer**. If the input is not a number, `InputMismatchException` is thrown.
- When done, call `keyboard.close()` .

```
Scanner keyboard = new Scanner(System.in);
System.out.println("enter some text");
String input = keyboard.next();
System.out.println(input);    // algorithms
input = keyboard.next();
System.out.println(input);    // in
input = keyboard.nextLine();
System.out.println(input);    //  computational terms
```

INPUT (KEYBOARD)

```
algorithms in computational terms
```

OUTPUT

```
enter some text
algorithms
in
 computational terms
```

`nextLine()` 's result starts with a **space**: the space after “in” is part of the line too.

USER INPUT · STEP-BY-STEP CODE TRACE

next() and nextLine(): Step by Step

Input buffer (empty)

(nothing typed yet)

The prompt is printed; the program waits for the user to enter a line.

Step 1 / 7

```
1 Scanner keyboard = new Scanner(System.in);
▶ System.out.println("enter some text");
3 String input = keyboard.next();
4 System.out.println(input);
5 input = keyboard.next();
6 System.out.println(input);
7 input = keyboard.nextLine();
8 System.out.println(input);
```

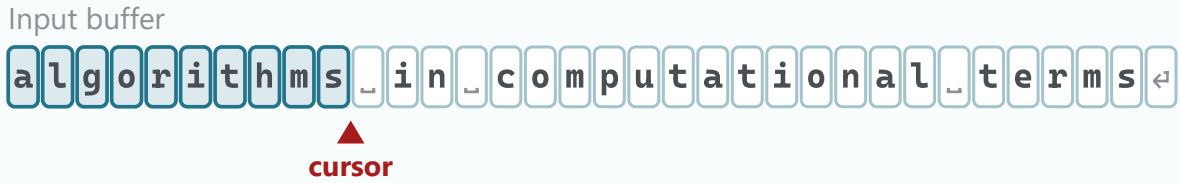
VARIABLES

—

OUTPUT

```
enter some text
```

next() and nextLine(): Step by Step



The user types the line and presses Enter (the buffer holds ⏎ too). `next()` reads up to the first space: **"algorithms"**.

Step 2 / 7

```
1 Scanner keyboard = new Scanner(System.in);
2 System.out.println("enter some text");
▶ String input = keyboard.next();
4 System.out.println(input);
5 input = keyboard.next();
6 System.out.println(input);
7 input = keyboard.nextLine();
8 System.out.println(input);
```

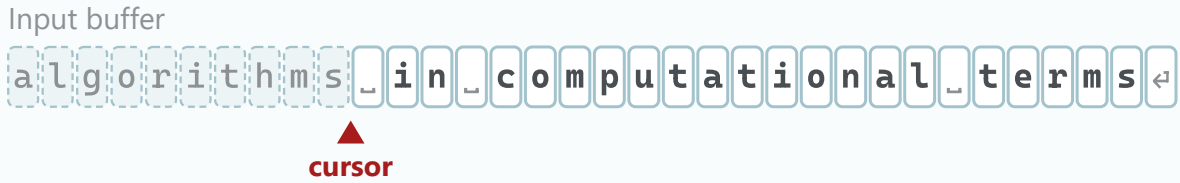
VARIABLES

`input = "algorithms"`

OUTPUT

```
enter some text
algorithms in computational terms
```

next() and nextLine(): Step by Step



The word is printed. The cursor waits before the space.

Step 3 / 7

```
1 Scanner keyboard = new Scanner(System.in);
2 System.out.println("enter some text");
3 String input = keyboard.next();
▶ System.out.println(input);
5 input = keyboard.next();
6 System.out.println(input);
7 input = keyboard.nextLine();
8 System.out.println(input);
```

VARIABLES

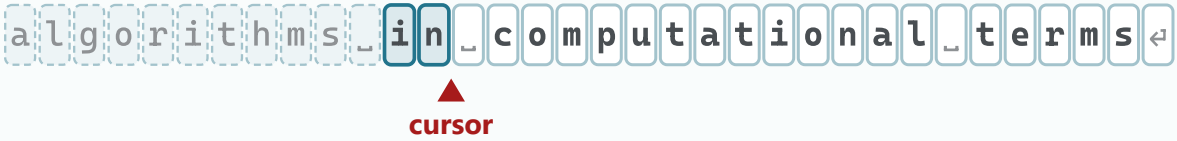
`input = "algorithms"`

OUTPUT

```
enter some text
algorithms in computational terms
algorithms
```

next() and nextLine(): Step by Step

Input buffer



`next()` first **skips** the space, then reads **"in"**.

Step 4 / 7

```
1 Scanner keyboard = new Scanner(System.in);
2 System.out.println("enter some text");
3 String input = keyboard.next();
4 System.out.println(input);
▶ input = keyboard.next();
6 System.out.println(input);
7 input = keyboard.nextLine();
8 System.out.println(input);
```

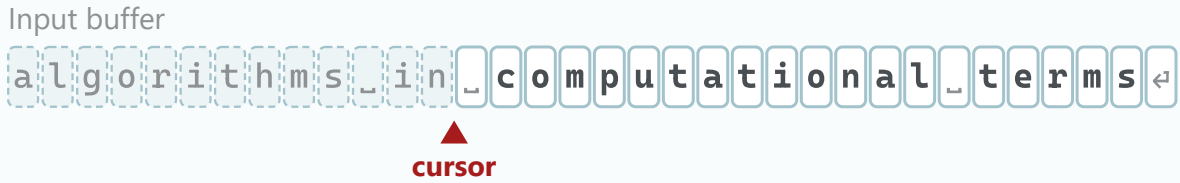
VARIABLES

`input = "in"`

OUTPUT

```
enter some text
algorithms in computational terms
algorithms
```

next() and nextLine(): Step by Step



Printed. The cursor is again before a space.

Step 5 / 7

```
1 Scanner keyboard = new Scanner(System.in);
2 System.out.println("enter some text");
3 String input = keyboard.next();
4 System.out.println(input);
5 input = keyboard.next();
▶ System.out.println(input);
7 input = keyboard.nextLine();
8 System.out.println(input);
```

VARIABLES

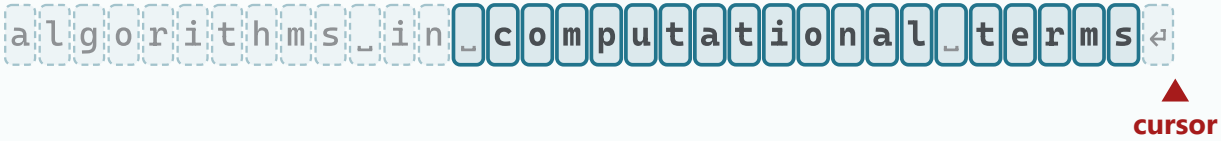
`input = "in"`

OUTPUT

```
enter some text
algorithms in computational terms
algorithms
in
```

next() and nextLine(): Step by Step

Input buffer



`nextLine()` does **not skip** the space: it reads to the end of the line and consumes ⏎: **" computational terms"**.

Step 6 / 7

```
1 Scanner keyboard = new Scanner(System.in);
2 System.out.println("enter some text");
3 String input = keyboard.next();
4 System.out.println(input);
5 input = keyboard.next();
6 System.out.println(input);
▶ input = keyboard.nextLine();
8 System.out.println(input);
```

VARIABLES

`input = " computational terms"`

OUTPUT

```
enter some text
algorithms in computational terms
algorithms
in
```

next() and nextLine(): Step by Step

Input buffer



Printed with the leading space. The buffer is now empty.

Step 7 / 7

```
1 Scanner keyboard = new Scanner(System.in);
2 System.out.println("enter some text");
3 String input = keyboard.next();
4 System.out.println(input);
5 input = keyboard.next();
6 System.out.println(input);
7 input = keyboard.nextLine();
▶ System.out.println(input);
```

VARIABLES

`input = " computational terms"`

OUTPUT

```
enter some text
algorithms in computational terms
algorithms
in
 computational terms
```

USER INPUT

The nextLine() after nextInt() Trap

Wrong: the name is skipped without being asked.

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Age: ");
int age = keyboard.nextInt();
System.out.print("Name: ");
String name = keyboard.nextLine();
System.out.println("[" + name + "] " + age);
```

CONSOLE (INPUT: 19 ⏎ AYŞE ⏎)

```
Age: 19
Name: [] 19
```

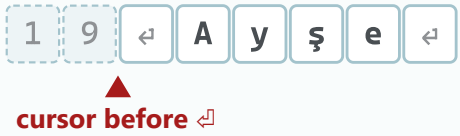
Right: consume the leftover newline.

```
int age = keyboard.nextInt();
keyboard.nextLine();          // drop ⏎
System.out.print("Name: ");
String name = keyboard.nextLine();
```

CONSOLE

```
Age: 19
Name: Ayşe
[Ayşe] 19
```

Buffer after nextInt()



`nextInt()` takes only `19`. The `nextLine()` right after it reads the **empty** text up to `⏎`. Alternative: read everything with `nextLine()` and convert with `Integer.parseInt`.

COMPARISON

Comparing with ==

- `==` compares the **references** (memory addresses) of objects, **not their contents**.
- If two variables point to the **same object**, it returns `true`; for different objects it returns `false`, even if the contents match.
- `str1 == str2` being `true` is no accident but a result of the **string pool**.

Bottom line: `==` **cannot compare** string contents; compare contents with `equals()`.

```
String str1 = "Giresun";
String str2 = "Giresun";
String str3 = new String("Giresun");
System.out.println(str1 == str2);         // true
System.out.println(str1 == "Giresun");    // true
System.out.println(str1 == str3);         // false
System.out.println(str3 == "Giresun");    // false
```

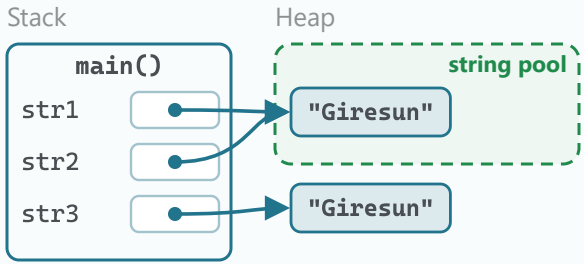
OUTPUT

```
true
true
false
false
```

COMPARISON

The String Pool

- Constants in source code such as `"giresun"` are **string literals**.
- Literals are stored in a special region of the heap, the **string pool**: literals with the same content **share one object**. This saves memory and time (it is safe because strings are immutable).
- `new String(...)` bypasses the pool: each `new` **creates a new object**.
- So `==` gives `true` for equal **literals** but `false` for equal `new` objects.
- A string built with `+` at run time is a new object too; a compile-time constant like `"Gire" + "sun"` goes to the pool. `s.intern()` returns the pooled copy.

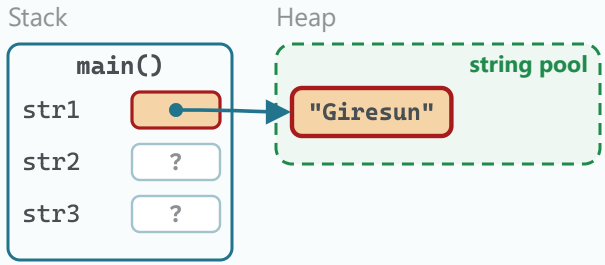


Two literals share one object; `new` creates a separate one.

COMPARISON · STEP-BY-STEP CODE TRACE

The String Pool: Step by Step

```
▶ String str1 = "Giresun";
2 String str2 = "Giresun";
3 String str3 = new String("Giresun");
4 System.out.println(str1 == str2);
5 System.out.println(str1 == "Giresun");
6 System.out.println(str1 == str3);
7 System.out.println(str3 == "Giresun");
```



The pool holds an object for the literal **"Giresun"**; `str1` points to it.

Step 1 / 7

VARIABLES

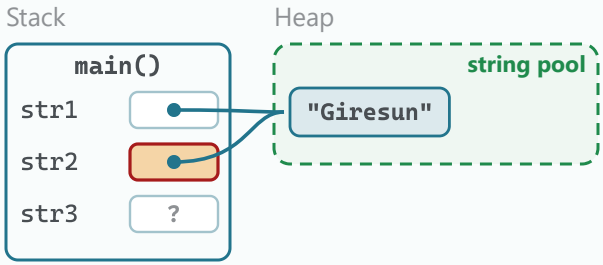
`str1 = →#1` `str2 = ?` `str3 = ?`

OUTPUT

COMPARISON · STEP-BY-STEP CODE TRACE

The String Pool: Step by Step

```
1 String str1 = "Giresun";
▶ String str2 = "Giresun";
3 String str3 = new String("Giresun");
4 System.out.println(str1 == str2);
5 System.out.println(str1 == "Giresun");
6 System.out.println(str1 == str3);
7 System.out.println(str3 == "Giresun");
```



The same literal is reused: no new object is **created**, `str2` also points to it.

Step 2 / 7

VARIABLES

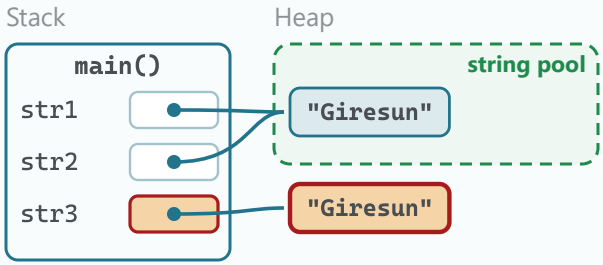
`str1 = →#1` `str2 = →#1` `str3 = ?`

OUTPUT

COMPARISON · STEP-BY-STEP CODE TRACE

The String Pool: Step by Step

```
1 String str1 = "Giresun";
2 String str2 = "Giresun";
▶ String str3 = new String("Giresun");
4 System.out.println(str1 == str2);
5 System.out.println(str1 == "Giresun");
6 System.out.println(str1 == str3);
7 System.out.println(str3 == "Giresun");
```



`new` always creates a **new** object outside the pool (same content).

Step 3 / 7

VARIABLES

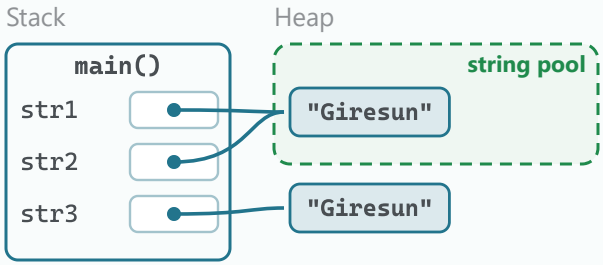
`str1 = →#1` `str2 = →#1` `str3 = →#2`

OUTPUT

COMPARISON · STEP-BY-STEP CODE TRACE

The String Pool: Step by Step

```
1 String str1 = "Giresun";
2 String str2 = "Giresun";
3 String str3 = new String("Giresun");
▶ System.out.println(str1 == str2);
5 System.out.println(str1 == "Giresun");
6 System.out.println(str1 == str3);
7 System.out.println(str3 == "Giresun");
```



`str1` and `str2` are the same object (#1) → **true**.

Step 4 / 7

VARIABLES

`str1 = →#1` `str2 = →#1` `str3 = →#2`

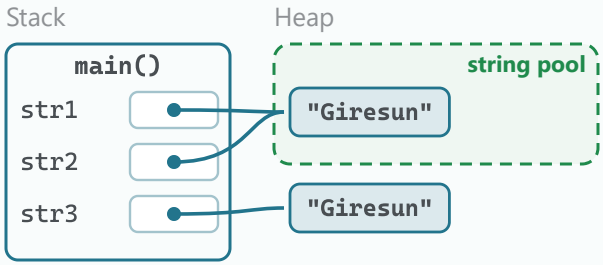
OUTPUT

```
true
```

COMPARISON · STEP-BY-STEP CODE TRACE

The String Pool: Step by Step

```
1 String str1 = "Giresun";
2 String str2 = "Giresun";
3 String str3 = new String("Giresun");
4 System.out.println(str1 == str2);
▶ System.out.println(str1 == "Giresun");
6 System.out.println(str1 == str3);
7 System.out.println(str3 == "Giresun");
```



`"Giresun"` is also the pooled #1 → **true**.

Step 5 / 7

VARIABLES

`str1 = →#1` `str2 = →#1` `str3 = →#2`

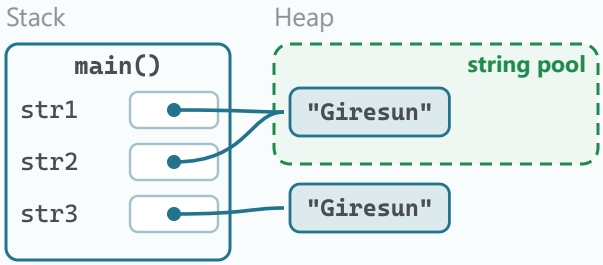
OUTPUT

```
true
true
```

COMPARISON · STEP-BY-STEP CODE TRACE

The String Pool: Step by Step

```
1 String str1 = "Giresun";
2 String str2 = "Giresun";
3 String str3 = new String("Giresun");
4 System.out.println(str1 == str2);
5 System.out.println(str1 == "Giresun");
▶ System.out.println(str1 == str3);
7 System.out.println(str3 == "Giresun");
```



#1 and #2 are different objects (same content) → **false**.

Step 6 / 7

VARIABLES

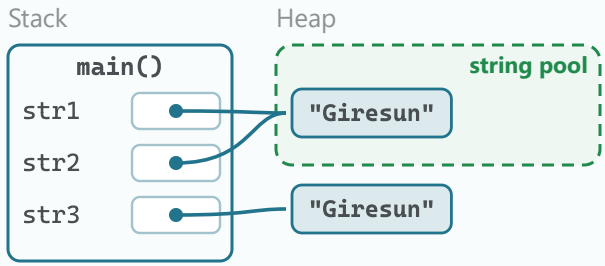
`str1 = →#1` `str2 = →#1` `str3 = →#2`

OUTPUT

```
true
true
false
```

The String Pool: Step by Step

```
1 String str1 = "Giresun";
2 String str2 = "Giresun";
3 String str3 = new String("Giresun");
4 System.out.println(str1 == str2);
5 System.out.println(str1 == "Giresun");
6 System.out.println(str1 == str3);
▶ System.out.println(str3 == "Giresun");
```



#2 differs from the pooled #1 → **false**.

Step 7 / 7

VARIABLES

`str1 = →#1` `str2 = →#1` `str3 = →#2`

OUTPUT

```
true
true
false
false
```

COMPARISON

Comparing Contents with equals()

```
String str1 = "Giresun";
String str2 = "Giresun";
String str3 = new String("Giresun");
System.out.println(str1.equals(str2));
System.out.println(str1.equals("Giresun"));
System.out.println(str1.equals(str3));
System.out.println(str3.equals("Giresun"));
System.out.println(str3.intern() == str1);
String a = "Gire";
String b = a + "sun";  // new object
System.out.println(b == str1);
System.out.println(b.equals(str1));
```

OUTPUT

```
true
true
true
true
true
false
true
```

- `equals()` **compares contents** character by character: the first four are `true`.
- How the object was made (literal, `new`, concatenation, `Scanner`) does not matter.
- User input is created at run time: `input == "yes"` is almost always `false`.

```
String answer = null;
System.out.println("yes".equals(answer)); // false
System.out.println(answer.equals("yes")); // error!
```

```
Exception in thread "main"
java.lang.NullPointerException:
    Cannot invoke "String.equals(Object)" because
"answer" is null
```

Putting the literal first (`"yes".equals(answer)`) is safe against `null`.

COMPARISON

Comparison Methods

Method	Returns	Description
`equals(str)`	`boolean`	Exactly the same contents?
`equalsIgnoreCase(str)`	`boolean`	The same, ignoring case?
`startsWith(str)`	`boolean`	Does the string **start** with the given string?
`endsWith(str)`	`boolean`	Does the string **end** with the given string?
`contains(str)`	`boolean`	Does the given string occur **inside** it?
`isEmpty()` / `isBlank()`	`boolean`	Length 0? / Only whitespace? (`isBlank` : Java 11+)
`compareTo(str)`	`int`	Lexicographic order: negative (before), 0 (equal), positive (after).
`compareToIgnoreCase(str)`	`int`	Case-insensitive `compareTo` .

All are **case-sensitive** (except those with *IgnoreCase* in the name), and with a `null` argument, `equals` / `equalsIgnoreCase` simply return `false` .

Comparison Methods: Example

```
String s = "Giresun Üni.";
System.out.println(s.startsWith("Gir"));                  // true
System.out.println(s.startsWith("Gır"));                  // false
System.out.println(s.endsWith("Üni."));                   // true
System.out.println(s.endsWith("Üni"));                    // false
System.out.println(s.equalsIgnoreCase("Giresun Üni."));   // true
System.out.println(s.equalsIgnoreCase("Gıresun üni."));   // true
System.out.println(s.equalsIgnoreCase("GIresun Uni."));   // false
System.out.println(s.contains("Üni"));                    // true
```

- The comments show the program's actual output.
- `"Gır"` : `ı` and `i` are different characters → `false`.
- `"Üni"` : the string ends with a period → `false`.
- `equalsIgnoreCase` compares after converting characters to uppercase: `ı` and `i` both become `I` → `"Gıresun üni."` counts as equal.
- `"GIresun Uni."` : `U` and `Ü` are different letters → `false`.



COMPARISON

compareTo: Lexicographic Order

```
System.out.println("egg".compareTo("apple"));
System.out.println("apple".compareTo("egg"));
System.out.println("egg".compareTo("egg"));
System.out.println("Egg".compareTo("egg"));
System.out.println("Egg".compareToIgnoreCase("egg"));
System.out.println("do".compareTo("door"));
System.out.println("çay".compareTo("dut"));
```

OUTPUT

```
4
-4
0
-32
0
-2
131
```

- `a.compareTo(b)`: `a` comes first → **negative**; equal → **0**; after → **positive**.
- The Unicode codes of the first differing characters are subtracted: `'e' - 'a'` = 101 − 97 = **4**.
- `'E' - 'e'` = 69 − 101 = **−32**: uppercase letters come before lowercase.
- If one is a prefix of the other, the length difference is returned: `"do"` vs `"door"` → 2 − 4 = **−2**.
- `'ç'` (231) > `'d'` (100): `compareTo` does **not know** Turkish alphabetical order, so "çay" comes after "dut". For correct Turkish sorting use `java.text.Collator`.
- Used for sorting (e.g. `Arrays.sort`) and in tests like `if (a.compareTo(b) < 0)`. `<`, `>` do not work on strings.

compareTo: Up to the First Difference

```
▶ String a = "case", b = "cave";
2 int r1 = a.compareTo(b);
3 int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```

case · cave · car · cargo · Zeki · ali

`compareTo` compares two strings character by character from the start.

Step 1 / 9

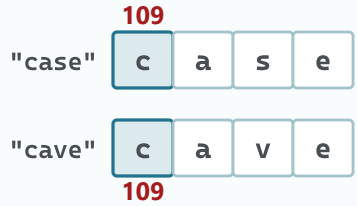
VARIABLES

—

OUTPUT

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
▶ int r1 = a.compareTo(b);
3 int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```



k = 0: 'c' (99) and 'c' (99) are equal; continue.

Step 2 / 9

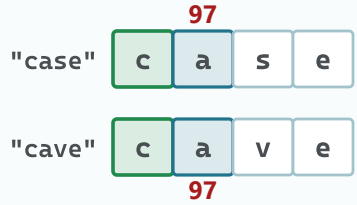
VARIABLES

k = 0

OUTPUT

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
▶ int r1 = a.compareTo(b);
3 int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```



k = 1: 'a' and 'a' are equal.

Step 3 / 9

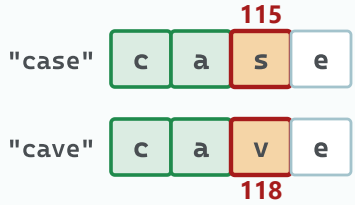
VARIABLES

k = 1

OUTPUT

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
▶ int r1 = a.compareTo(b);
3 int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```



k = 2: 's' (115) and 'v' (118) differ: the comparison stops here.

Step 4 / 9

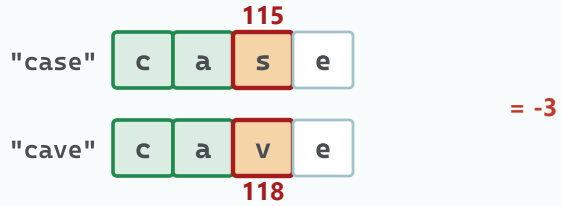
VARIABLES

k = 2

OUTPUT

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
▶ int r1 = a.compareTo(b);
3 int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```



The result is the code difference: 115 − 118 = **−3**.
Negative: "case" comes first.

Step 5 / 9

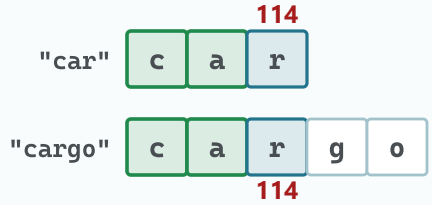
VARIABLES

`r1 = -3`

OUTPUT

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
2 int r1 = a.compareTo(b);
▶ int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```



"car" and "cargo": the first 3 characters match and the shorter one runs out.

Step 6 / 9

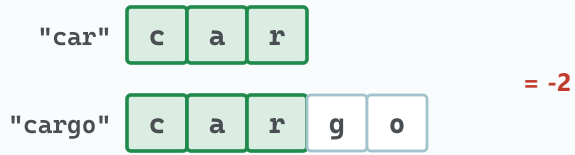
VARIABLES

`r1 = -3`

OUTPUT

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
2 int r1 = a.compareTo(b);
▶ int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```



One is a prefix of the other: the length difference is returned, 3 − 5 = **−2**.

Step 7 / 9

VARIABLES

`r1 = -3` `r2 = -2`

OUTPUT

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
2 int r1 = a.compareTo(b);
3 int r2 = "car".compareTo("cargo");
▶ int r3 = "Zeki".compareTo("ali");
5 System.out.println(r1 + " " + r2 + " " + r3);
```



The first characters differ: 'Z' (90) − 'a' (97) = **−7**.
Uppercase comes before lowercase: even Z precedes a!

Step 8 / 9

VARIABLES

`r1 = -3` `r2 = -2` `r3 = -7`

OUTPUT

COMPARISON · STEP-BY-STEP CODE TRACE

compareTo: Up to the First Difference

```
1 String a = "case", b = "cave";
2 int r1 = a.compareTo(b);
3 int r2 = "car".compareTo("cargo");
4 int r3 = "Zeki".compareTo("ali");
▶ System.out.println(r1 + " " + r2 + " " + r3);
```

−3 −2 −7

`−3 −2 −7` is printed.

Step 9 / 9

VARIABLES

`r1 = −3` `r2 = −2` `r3 = −7`

OUTPUT

```
-3 -2 -7
```

FORMATTED OUTPUT

The printf Method

- Usage: `System.out.printf("format string", args);`
- The format string holds **format specifiers**; the arguments fill them in order.
- `printf` adds **no** newline: append `\n` or the portable `%n`.
- Width: `%Wd` → W characters, **right-aligned**; `%-Wd` → W characters, **left-aligned**.
- Precision: `%.2f` → 2 decimal places (rounded); `%8.2f` → 8 characters in total.
- A wider value is not cut; the field grows.

Specifier	Type	Example → output
`%d`	Integer	42 → `42`
`%f`	Decimal (`float` , `double`)	`%.1f` : 3.14159 → `3,1`
`%s`	String	`%5s` : "ab" → `␣␣␣ab`
`%c`	Character	`'Z'` → `Z`
`%b`	Boolean	`true` → `true`
`%n`	Newline	—

The decimal separator depends on the **locale**: in tr-TR it is `3,1`, with `Locale.US` it is `3.1`.



printf: Example

```
int i = 3;
float f = -17;
String s = "java";

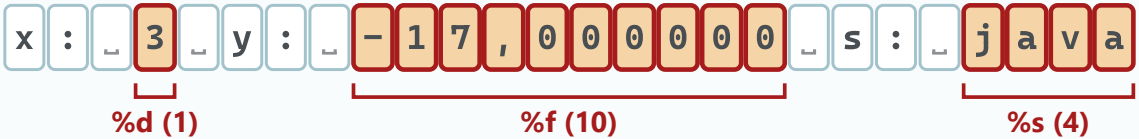
System.out.printf("x: %d y: %f s: %s\n", i, f, s);
System.out.printf("x: %4d y: %5.2f s: %6s\n", i, f, s);
System.out.printf("x: %-4d y: %-7.2f s: %-6s\n", i, f, s);
System.out.printf("x: %-4d y: %-6.3f s: %-6s\n", i, f, s);
```

OUTPUT (TR-TR; TRAILING SPACES SHOWN AS ·)

```
x: 3 y: -17,000000 s: java
x:    3 y: -17,00 s:   java
x: 3    y: -17,00  s: java··
x: 3    y: -17,000 s: java··
```

- `%f` prints 6 decimal places by default: `-17,000000`. Under a Turkish locale the decimal separator is a **comma**.
- `%5.2f` : `-17,00` is already 6 characters, so width 5 has no effect. `%-6.3f` : `-17,000` is 7 characters; the field grows.
- `%6s` → `␣␣java` (right-aligned); `%-6s` → `java␣␣` (left-aligned, padded on the right).

Width and Precision, Character by Character



Without a width, each value takes exactly its own length; `%f` prints 6 decimal places.

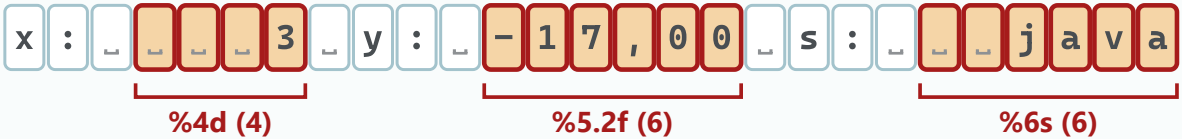
Step 1 / 4

```
1 int i = 3;
2 float f = -17;
3 String s = "java";
▶ System.out.printf("x: %d y: %f s: %s\n", i, f, s);
5 System.out.printf("x: %4d y: %5.2f s: %6s\n", i, f, s);
6 System.out.printf("x: %-4d y: %-7.2f s: %-6s\n", i, f, s);
7 System.out.printf("x: %-4d y: %-6.3f s: %-6s\n", i, f, s);
```

OUTPUT

```
x: 3 y: -17,000000 s: java
```

Width and Precision, Character by Character



`%4d` and `%6s` **right-align** the value, padding the left with spaces. `-17,00` is 6 characters: it does not fit width 5, so the field grows.

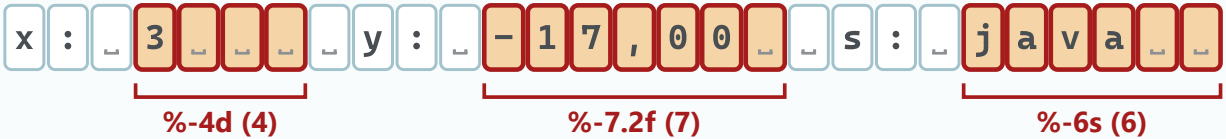
Step 2 / 4

```
1 int i = 3;
2 float f = -17;
3 String s = "java";
4 System.out.printf("x: %d y: %f s: %s\n", i, f, s);
▶ System.out.printf("x: %4d y: %5.2f s: %6s\n", i, f, s);
6 System.out.printf("x: %-4d y: %-7.2f s: %-6s\n", i, f, s);
7 System.out.printf("x: %-4d y: %-6.3f s: %-6s\n", i, f, s);
```

OUTPUT

```
x: 3 y: -17,000000 s: java
x:    3 y: -17,00 s:   java
```

Width and Precision, Character by Character



The minus sign **left-aligns**: spaces go to the right of the value, so columns line up neatly.

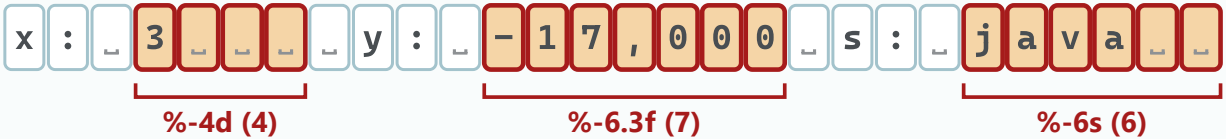
Step 3 / 4

```
1 int i = 3;
2 float f = -17;
3 String s = "java";
4 System.out.printf("x: %d y: %f s: %s\n", i, f, s);
5 System.out.printf("x: %4d y: %5.2f s: %6s\n", i, f, s);
▶ System.out.printf("x: %-4d y: %-7.2f s: %-6s\n", i, f, s);
7 System.out.printf("x: %-4d y: %-6.3f s: %-6s\n", i, f, s);
```

OUTPUT

```
x: 3 y: -17,000000 s: java
x:    3 y: -17,00 s:   java
x: 3    y: -17,00  s: java
```

Width and Precision, Character by Character



`%-6.3f`: 3 decimals → `-17,000` 7 characters; wider than 6, so no padding.

Step 4 / 4

```
1 int i = 3;
2 float f = -17;
3 String s = "java";
4 System.out.printf("x: %d y: %f s: %s\n", i, f, s);
5 System.out.printf("x: %4d y: %5.2f s: %6s\n", i, f, s);
6 System.out.printf("x: %-4d y: %-7.2f s: %-6s\n", i, f, s);
▶ System.out.printf("x: %-4d y: %-6.3f s: %-6s\n", i, f, s);
```

OUTPUT

```
x: 3 y: -17,000000 s: java
x:    3 y: -17,00 s:   java
x: 3    y: -17,00  s: java
x: 3    y: -17,000 s: java
```

More Formats and String.format

```
System.out.printf("|%5s|%-5s|%05d|%.1f|%n", "ab", "ab", 42, 3.14159);
System.out.printf("%c %b %e %x %%%n", 'Z', true, 1234.5, 255);
String msg = String.format("%s: %d points", "Ayşe", 85);
System.out.println(msg);
System.out.printf("%,d%n", 1234567);
System.out.printf(Locale.US, "%.2f%n", 2.5);       // import java.util.Locale;
System.out.println("%s = %d".formatted("x", 7));   // Java 15+
```

OUTPUT

```
|   ab|ab   |00042|3,1|
Z true 1,234500e+03 ff %
Ayşe: 85 points
1.234.567
2.50
x = 7
```

- `%05d` pads with leading zeros, `%,d` adds a thousands separator (tr-TR: period), `%e` scientific, `%x` hexadecimal, `%%` is a percent sign.
- `String.format(...)` uses the same rules but, instead of printing, **returns the formatted string**.
- The first argument can be a `Locale`: `Locale.US` gives a period as the decimal separator.
- A type mismatch is a runtime error: `printf("%d", 3.5)` → `IllegalFormatConversionException`.

Palindrome Check: Step by Step

```
▶ String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
5   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.

text

K	a	y	a	k
0	1	2	3	4

rev "" *(empty)*

The text entered by the user: `"Kayak"`.

Step 1 / 11

VARIABLES

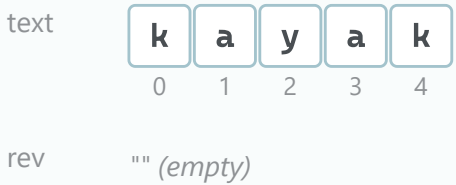
`text = "Kayak"`

OUTPUT

Palindrome Check: Step by Step

```
1 String text = "Kayak";
▶ text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
5   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.



`toLowerCase()` converts it to lowercase (the new object is assigned to `text`).

Step 2 / 11

VARIABLES

`text = "kayak"`

OUTPUT

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
▶ String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
5   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.

text

k	a	y	a	k
0	1	2	3	4

rev "" *(empty)*

An empty string is prepared for the result.

Step 3 / 11

VARIABLES

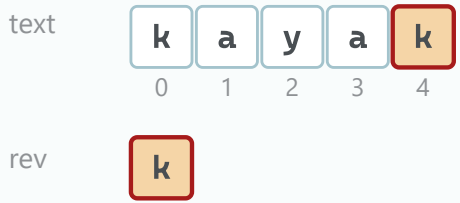
`text = "kayak"` `rev = ""`

OUTPUT

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
▶   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.



`i = 4` : `charAt(4)` = 'k' is appended → **"k"**.

Step 4 / 11

VARIABLES

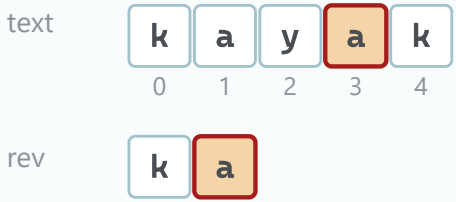
`text = "kayak"` `rev = "k"` `i = 4`

OUTPUT

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
▶   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.



`i = 3` : `charAt(3)` = 'a' is appended → **"ka"**.

Step 5 / 11

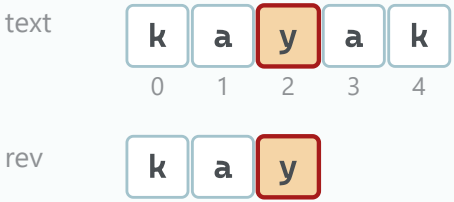
VARIABLES

`text = "kayak"` `rev = "ka"` `i = 3`

OUTPUT

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
▶   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```



`i = 2` : `charAt(2)` = 'y' is appended → **"kay"**.

Step 6 / 11

VARIABLES

`text = "kayak"` `rev = "kay"` `i = 2`

OUTPUT

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
▶   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```



`i = 1` : `charAt(1)` = 'a' is appended → **"kaya"**.

Step 7 / 11

VARIABLES

`text = "kayak"` `rev = "kaya"` `i = 1`

OUTPUT

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
▶   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```



`i = 0` : `charAt(0)` = 'k' is appended → **"kayak"**.

Step 8 / 11

VARIABLES

`text = "kayak"` `rev = "kayak"` `i = 0`

OUTPUT

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
▶ for (int i = text.length() - 1; i >= 0; i--) {
5   rev += text.charAt(i);
6 }
7 System.out.println(rev);
8 System.out.println(text.equals(rev));
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.



`i = -1` : `-1 >= 0` is false; the loop ends.

Step 9 / 11

VARIABLES

`text = "kayak"` `rev = "kayak"` `i = -1`

OUTPUT

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
5   rev += text.charAt(i);
6 }
▶ System.out.println(rev);
8 System.out.println(text.equals(rev));
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; "A man, a plan, a canal: Panama" is a palindrome once cleaned.



The reversed text is printed.

Step 10 / 11

VARIABLES

`text = "kayak"` `rev = "kayak"`

OUTPUT

```
kayak
```

Palindrome Check: Step by Step

```
1 String text = "Kayak";
2 text = text.toLowerCase();
3 String rev = "";
4 for (int i = text.length() - 1; i >= 0; i--) {
5   rev += text.charAt(i);
6 }
7 System.out.println(rev);
▶ System.out.println(text.equals(rev));
```



Same contents: `equals` → **true**, so “kayak” is a palindrome.

Step 11 / 11

VARIABLES

`text = "kayak"` `rev = "kayak"`

OUTPUT

```
kayak
true
```

A palindrome reads the same backward (kayak, level, noon). For sentences, first remove non-letters: `text.replaceAll("[^\\p{L}]", "")` (only letters remain). A shortcut for reversing: `new StringBuilder(text).reverse().toString()`; “A man, a plan, a canal: Panama” is a palindrome once cleaned.

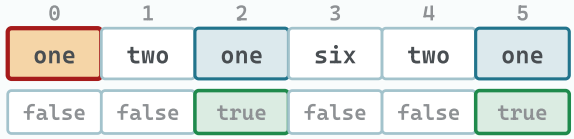
Application: Word Frequency

```
String text = "One two one six TWO one";
String[] words = text.toLowerCase().split(" ");
boolean[] counted = new boolean[words.length];
for (int i = 0; i < words.length; i++) {
  if (counted[i]) {
    continue; // already counted
  }
  int freq = 1;
  for (int j = i + 1; j < words.length; j++) {
    if (words[i].equals(words[j])) {
      freq++;
      counted[j] = true;
    }
  }
  System.out.println(words[i] + ": " + freq);
}
```

OUTPUT

```
one: 3
two: 2
six: 1
```

- `toLowerCase()` makes “One/one” and “TWO/two” the same word.
- `split(" ")` splits the text into words.
- Words are compared with **`equals`**: `split` returns separate objects, so `==` gives wrong results.
- The `counted` array stops a word being reported twice.



After i = 0: "one" 3 times, copies flagged

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
▶ 2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
5    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
8      if (k[i].equals(k[j])) {
9        freq++;
10       counted[j] = true;
11     }
12   }
13   System.out.println(k[i] + ": " + freq);
14 }
```

	0	1	2	3	4
k	apple	pear	apple	plum	pear

Lowercased and split on spaces: an array of 5 words. "Pear" and "PEAR" are now the same.

Step 1 / 9

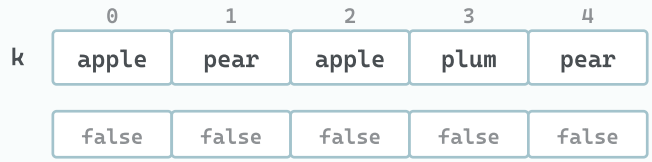
VARIABLES

—

OUTPUT

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
▶  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
5    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
8      if (k[i].equals(k[j])) {
9        freq++;
10       counted[j] = true;
11     }
12   }
13   System.out.println(k[i] + ": " + freq);
14 }
```



`counted` : one flag per word, all `false` .

Step 2 / 9

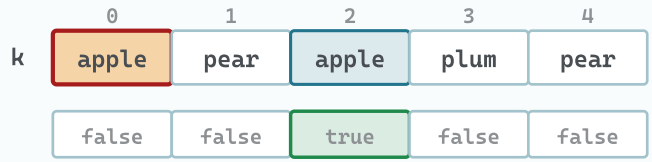
VARIABLES

—

OUTPUT

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
5    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
▶      if (k[i].equals(k[j])) {
▶        freq++;
▶        counted[j] = true;
11     }
12   }
13   System.out.println(k[i] + ": " + freq);
14 }
```



i = 0 (apple): the inner loop scans the words to the right with `equals`; j = 2 matches → freq 2, `counted[2] = true`.

Step 3 / 9

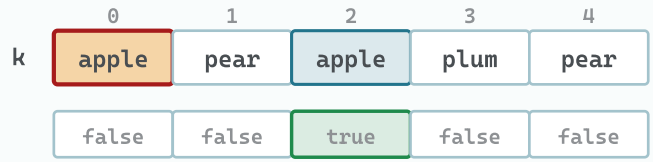
VARIABLES

`i = 0` `freq = 2`

OUTPUT

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
5    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
8      if (k[i].equals(k[j])) {
9        freq++;
10       counted[j] = true;
11     }
12   }
▶    System.out.println(k[i] + ": " + freq);
14 }
```



`apple: 2` is printed.

Step 4 / 9

VARIABLES

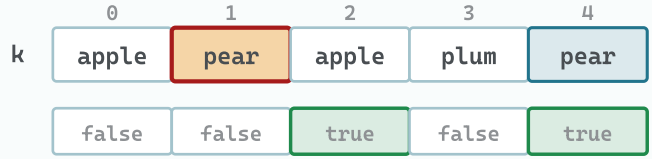
`i = 0` `freq = 2`

OUTPUT

```
apple: 2
```

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
5    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
▶      if (k[i].equals(k[j])) {
▶        freq++;
▶        counted[j] = true;
11     }
12   }
13   System.out.println(k[i] + ": " + freq);
14 }
```



i = 1 (pear): j = 4 matches → freq 2, `counted[4] = true`.

Step 5 / 9

VARIABLES

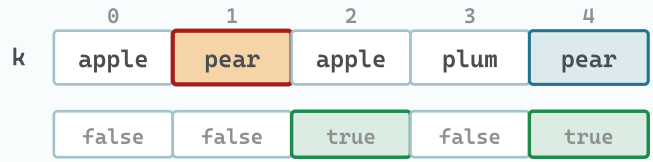
`i = 1` `freq = 2`

OUTPUT

```
apple: 2
```

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
5    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
8      if (k[i].equals(k[j])) {
9        freq++;
10       counted[j] = true;
11     }
12   }
▶    System.out.println(k[i] + ": " + freq);
14 }
```



`pear: 2` is printed.

Step 6 / 9

VARIABLES

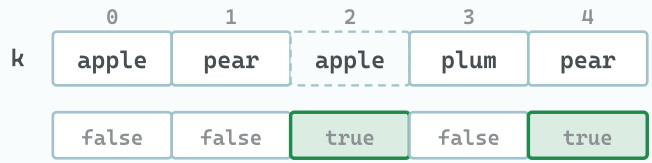
`i = 1` `freq = 2`

OUTPUT

```
apple: 2
pear: 2
```

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
▶    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
8      if (k[i].equals(k[j])) {
9        freq++;
10       counted[j] = true;
11     }
12   }
13   System.out.println(k[i] + ": " + freq);
14 }
```



i = 2: flag is true, this apple was already counted → `continue`.

Step 7 / 9

VARIABLES

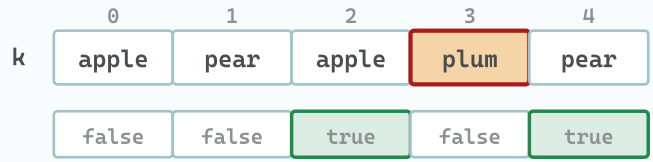
`i = 2`

OUTPUT

```
apple: 2
pear: 2
```

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
5    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
8      if (k[i].equals(k[j])) {
9        freq++;
10       counted[j] = true;
11     }
12   }
▶    System.out.println(k[i] + ": " + freq);
14 }
```



i = 3 (plum): no match, freq 1.

Step 8 / 9

VARIABLES

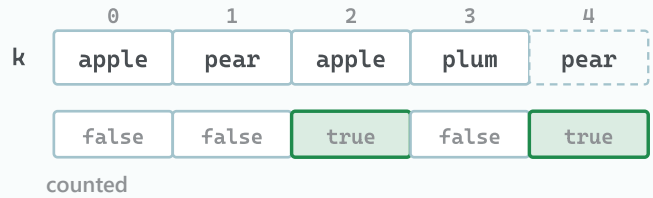
`i = 3` `freq = 1`

OUTPUT

```
apple: 2
pear: 2
plum: 1
```

Word Frequency: Counting with Flags

```
1  String text = "apple Pear apple plum PEAR";
2  String[] k = text.toLowerCase().split(" ");
3  boolean[] counted = new boolean[k.length];
4  for (int i = 0; i < k.length; i++) {
▶    if (counted[i]) continue;
6    int freq = 1;
7    for (int j = i + 1; j < k.length; j++) {
8      if (k[i].equals(k[j])) {
9        freq++;
10       counted[j] = true;
11     }
12   }
13   System.out.println(k[i] + ": " + freq);
14 }
```



i = 4: already counted, skipped. The loop ends.

Step 9 / 9

VARIABLES

`i = 4`

OUTPUT

```
apple: 2
pear: 2
plum: 1
```

Summary

Concept	Key idea
`String`	An **immutable** object holding a character sequence; created from a literal (`"..."`) or with `new` .
Index	0 … `length() - 1` ; `charAt(i)` returns a `char` .
`+`	Left to right: concatenation once one side is a string; use parentheses to add numbers.
Methods	`indexOf` (−1 = not found), `substring(a, b)` (b excluded), `toUpperCase` , `replace` , `split` , `trim` … all return new values.
`StringBuilder`	Mutable buffer for heavy concatenation in loops.
`Scanner`	`next()` a word, `nextLine()` a line; `nextInt()` leaves ⏎ in the buffer.
`==` / `equals`	Same object? / Same contents? For strings, always use `equals` .
String pool	Equal literals share one object; `new` and runtime concatenation create new objects.
Comparison	`equalsIgnoreCase` , `startsWith` , `endsWith` , `contains` , `compareTo` (Unicode order).
`printf`	`%d %f %s` ; `%Wd` right-, `%-Wd` left-aligned; `%.2f` precision; separators depend on the locale.

SUMMARY

Common Mistakes (1): Compiler Errors

A string in single quotes

```
error: unclosed character literal
```

`String s = 'Hello';` Single quotes are for one character only; strings use double quotes.

Assigning a string to a char

```
error: incompatible types: String cannot be converted to char
```

`char c = "a";` → use `char c = 'a';` or `"a".charAt(0)`.

length without parentheses

```
error: cannot find symbol
  symbol:   variable length
```

For strings, `s.length()` is a method. The reverse is an error too: for an array, `d.length()` → `symbol: method length()`.

string in lowercase

```
error: cannot find symbol
  symbol:   class string
```

The class name is `String`; Java is case-sensitive.

Assigning a string to a number

```
error: incompatible types: String cannot be converted to int
```

`int n = "42";` → use `Integer.parseInt("42")`.

Changing a character in place

```
error: unexpected type
  required: variable
```

`s.charAt(0) = 'j';` is illegal: strings are immutable. Fix: build a new string or use `StringBuilder.setCharAt`.

SUMMARY

Common Mistakes (2): Logic and Runtime

Comparing contents with ==

```
if (input == "yes")   // compiles but wrong
```

User input is a new object; the condition is `false`. Correct: `"yes".equals(input)`.

Forgetting to assign the result

```
s.toUpperCase();   // s unchanged
```

Strings are immutable; write `s = s.toUpperCase();`.

Taking length() as the last index

```
StringIndexOutOfBoundsException: Index 4 out of bounds for length 4
```

The last character is `s.charAt(s.length() - 1)`; the loop condition is `i < s.length()`.

nextLine after nextInt

```
String name = keyboard.nextLine();   // ""
```

The leftover ⏎ is read. Fix: add an extra `keyboard.nextLine();`.

Calling a method on a null string

```
NullPointerException: Cannot invoke "String.equals(Object)"
```

With the literal first, there is no error: `"yes".equals(answer)`.

Thinking substring's end is included

```
s.substring(5, 12)   // 7 chars
```

The end index is **exclusive**: length = end − start.

SUMMARY

Worked Problems (1)

Problem 1. What does it print?

```
String s = "Karadeniz";
s.toUpperCase();
String t = s.substring(2, 5);
System.out.println(t + " " + t.length());
System.out.println(s.indexOf("de"));
System.out.println(s.indexOf("x"));
System.out.println(1 + 2 + "3" + 4 + 5);
```

Problem 2. What does it print?

```
String a = "code";
String b = "co" + "de";
String c = new String("code");
String d = c;
System.out.println(a == b);
System.out.println(a == c);
System.out.println(c == d);
System.out.println(a.equals(c));
```

▶ **Show answers**

SUMMARY

Worked Problems (2)

Problem 3. Write the output character by character (tr-TR).

```
String name = "Ali";
double grade = 72.456;
System.out.printf("[%-5s][%3d][%7.2f]%n", name, 7, grade);
```

Problem 4. What does it print?

```
public static String initials(String first, String last) {
  return first.charAt(0) + "." + last.charAt(0) + ".";
}
System.out.println(initials("Sercan", "Külcü"));
String e = "student@giresun.edu.tr";
int at = e.indexOf('@');
System.out.println(e.substring(0, at));
System.out.println(e.substring(at + 1));
```

▶ **Show answers**

SUMMARY

Exercises (1)

1. Write `reverse(String s)` : `reverse("Giresun")` → `"nuseriG"` .
2. `count(String s, char target)` : returns how many times the character occurs. `count("Karadeniz", 'a')` → ?
3. `capitalize(String word)` : first letter uppercase, the rest lowercase. `capitalize("jAVA")` → `"Java"` .

▶ **Show answers**

SUMMARY

Exercises (2)

4. `"Object oriented programming with Java"` : how many words? Find out in one line.
5. Write a program that counts the vowels in a word. `"Ordu"` → ?
6. Ask for the age, then the full name, and print “Name: …, Age: …” with `printf`, avoiding the `nextInt` trap.

▶ **Show answers**

References

- Oracle, *The Java Tutorials*: “Strings”, “Comparing Strings and Portions of Strings”, “The StringBuilder Class”, “Formatting Numeric Print Output”.
- Oracle, *Java SE 21 API Documentation*: `java.lang.String`, `java.lang.StringBuilder`, `java.util.Scanner`, `java.util.Formatter`, `java.util.Locale`.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition* (§3.10.5 String Literals, §15.18.1 String Concatenation Operator +).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson, 2017 (Chapter 14: Strings, Characters and Regular Expressions).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022 (Section 3.6: Strings).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 6: Loops and Strings).

The End

Chapter 3.4: Strings

Previous: Chapter 3.3: Arrays · Next: Chapter 3.5: Recursion