For the given method, provide the syntax, its purpose, its priority to use in problem solving, core examples contains what its takes, its return type and then print its value, then given method is "int length()"

Got it \_\_\_\_ Let's carefully break down the int length() method in Java String class the way you asked:

## 1. Syntax

int length()

## 2. Purpose

- The length() method is used to **find the total number of characters** present in a string (including spaces, digits, and special characters).
- It's very useful when solving problems involving string size checks, validations, loops, or conditions.

## 3. Priority in Problem Solving

- ✓ High Priority You will use it often in coding problems like:
  - Checking if a string is empty or not.
  - Iterating over characters in a string.
  - Validating input length (like password rules).

Substring operations that depend on size.

# 4. Core Examples

## **Example 1: Simple string**

```
public class Example1 {
   public static void main(String[] args) {
      String str = "Hello";
      int len = str.length(); // counts characters
      System.out.println("Length: " + len);
   }
}
```

#### **Output:**

Length: 5

## **Example 2: String with spaces**

```
public class Example2 {
    public static void main(String[] args) {
        String str = "Hello World";
        System.out.println("Length: " + str.length());
    }
}
```

#### **Output:**

Length: 11

## **Example 3: Empty string**

```
public class Example3 {
   public static void main(String[] args) {
      String str = "";
      System.out.println("Length: " + str.length());
```

```
}
}
```

#### **Output:**

Length: 0

## **Example 4: Useful in conditions**

```
public class Example4 {
   public static void main(String[] args) {
      String password = "abc123";

   if (password.length() < 8) {
      System.out.println("Password is too short!");
   } else {
      System.out.println("Password length is fine.");
   }
}</pre>
```

#### **Output:**

Password is too short!

# 5. What it takes & Return type

- Takes: X No arguments (it works directly on the string object).
- **Returns:** int → the number of characters in the string.

So the **length()** method is **simple but very powerful** and should be one of the first tools you use when working with strings.

Do you want me to also cover **comparison with length (array property)**, since beginners often get confused between array.length and string.length()?

# boolean isEmpty()

Got it \_\_\_\_ Let's break down the **boolean isEmpty()** method in **Java Strings** step by step in a beginner-friendly way:

## 1. Syntax

boolean isEmpty()

## 2. Purpose

- The isEmpty() method checks whether a given **String** has a **length of 0**.
- In simple words: it tells you if the string is **empty ("")**.

## 3. Priority to Use in Problem Solving

- Use isEmpty() when you need to quickly check if a string has no characters.
- Very common in validations, input checks, and string handling problems.
- If you want to check **only spaces**, use isBlank() (Java 11+) instead.

## 4. What it Takes

• isEmpty() takes no arguments.

## 5. Return Type

- It returns a boolean:
  - o true  $\rightarrow$  if the string length is 0.
  - $\circ$  false  $\rightarrow$  if the string has 1 or more characters.

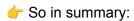
## 6. Core Examples

```
public class StringIsEmptyExample {
  public static void main(String[] args) {
    String str1 = "";  // empty string
    String str2 = "Hello"; // non-empty string
    String str3 = " ";  // string with a space (not empty)

    // Using isEmpty()
    System.out.println("str1 is empty? " + str1.isEmpty()); // true
    System.out.println("str2 is empty? " + str2.isEmpty()); // false
    System.out.println("str3 is empty? " + str3.isEmpty()); // false
    System.out.println("str3 is empty? " + str3.isEmpty()); // false
}
```

## **7. Output**

str1 is empty? true str2 is empty? false str3 is empty? false



- Takes nothing
- Returns boolean
- Checks if string length = 0
- Useful for input validations

Do you want me to also cover the **difference between isEmpty() and isBlank()** (since both are often confused)?

# boolean isBlank()

Alright \_\_\_ Let's break down the **boolean isBlank()** method in **Java** step by step in a beginner-friendly way.

#### 1. Syntax

boolean isBlank()

## • 2. Purpose

- The isBlank() method checks whether a string is empty or contains only whitespace characters (like spaces, tabs, or newline characters).
- It was introduced in Java 11.
- More powerful than isEmpty() because isEmpty() only checks if the string has no characters, while isBlank() also considers whitespace as "blank".

## 3. Priority to use in Problem Solving

- Use **isBlank()** when you need to validate **user input** or **filter strings** where whitespace should also be treated as empty.
- Higher priority compared to isEmpty() when handling real-world input (since users often enter spaces).

#### 4. Parameters It Takes

• X It does not take any arguments.

• It works directly on the **string object**.

## 5. Return Type

- Returns a boolean value:
  - true → if the string is empty ("") or contains only whitespace.
  - $\circ$  false  $\rightarrow$  otherwise.

## • 6. Examples

```
public class IsBlankExample {
  public static void main(String[] args) {
     String str1 = "";
                            // empty string
     String str2 = " ";
                            // only spaces
     String str3 = \n \n\t";
                            // newline + tab
     String str4 = "Hello";
                               // non-empty string
     String str5 = " Java "; // spaces + text
     System.out.println(str1.isBlank()); // true
     System.out.println(str2.isBlank()); // true
     System.out.println(str3.isBlank()); // true
     System.out.println(str4.isBlank()); // false
     System.out.println(str5.isBlank()); // false
}
```

## 7. Output

true

true

true

false

false

#### **✓** Summary:

Method: boolean isBlank()

• Takes: No arguments

• Returns: true or false

• When to use: Best for input validation where whitespace should be treated as empty.

Would you like me to also give you a **comparison table of isBlank() vs isEmpty()** so you know when to use which one?

# char charAt(int index)

Got it \_\_\_\_\_ Let's break down the char charAt(int index) method in Java in a beginner-friendly way.

## Method Details

#### 1. Syntax:

char variable = stringName.charAt(index);

#### 2. Purpose:

- The charAt() method is used to get a single character from a string at a specific index.
- Indexing starts at **0** (first character is at index Θ).

#### 3. Priority in Problem Solving:

• High priority when you want to access characters individually from a string.

• Useful in problems like palindrome checking, string reversal, pattern matching, substring creation, etc.

#### 4. Parameters (What it takes):

• Takes **one integer argument** (index) → the position of the character.

#### 5. Return Type (What it gives):

• Returns a **char** (single character).

## Core Examples

#### Example 1: Basic usage

```
public class Main {
   public static void main(String[] args) {
      String name = "Java";

      char firstChar = name.charAt(0); // 'J'
      char secondChar = name.charAt(1); // 'a'
      char lastChar = name.charAt(3); // 'a'

      System.out.println("First Character: " + firstChar);
      System.out.println("Second Character: " + secondChar);
      System.out.println("Last Character: " + lastChar);
    }
}
```

#### **Output:**

First Character: J Second Character: a Last Character: a

#### **Example 2: Loop through all characters**

```
public class Main {
  public static void main(String[] args) {
```

```
String word = "Hello";

for (int i = 0; i < word.length(); i++) {
    char c = word.charAt(i);
    System.out.println("Character at index " + i + ": " + c);
    }
}</pre>
```

#### **Output:**

Character at index 0: H Character at index 1: e Character at index 2: I Character at index 3: I Character at index 4: o

#### **Example 3: Using in Problem Solving (Palindrome check)**

```
public class Main {
   public static void main(String[] args) {
      String text = "madam";
      boolean isPalindrome = true;

   for (int i = 0; i < text.length() / 2; i++) {
      if (text.charAt(i) != text.charAt(text.length() - 1 - i)) {
         isPalindrome = false;
         break;
      }
   }
   System.out.println("Is Palindrome? " + isPalindrome);
   }
}</pre>
```

#### **Output:**

Is Palindrome? true

Do you want me to also explain **what happens if you give an invalid index** (like -1 or greater than string.length() - 1)?

# int codePointAt(int index)

Got it de Let's go step by step for the codePointAt(int index) method in Java.

## Method Details

#### 1. Syntax

int codePointAt(int index)

#### 2. Purpose

- This method **returns the Unicode code point** (an integer value) of the character at the specified index in a string.
- It is useful when dealing with **Unicode characters** beyond the basic ASCII set (like emojis, symbols, or characters from other languages).

## 3. Priority in Problem Solving

- **High priority** when working with:
  - Unicode characters (like emojis, Chinese, Hindi, etc.)
  - Converting characters to their integer code representation
  - Iterating or analyzing text with mixed languages/symbols
- For basic English alphabets (A–Z, a–z, 0–9), it mostly behaves like charAt(index) but returns the Unicode integer value instead of the char.

## Core Examples

#### **Example 1: Basic English Characters**

```
public class Example1 {
    public static void main(String[] args) {
        String str = "Java";

        // Takes -> index (0-based position)
        // Returns -> int (Unicode code point)
        int codePoint = str.codePointAt(0);

        System.out.println("Character at index 0: " + str.charAt(0));
        System.out.println("Code point at index 0: " + codePoint);
    }
}
```

## **Output:**

```
Character at index 0: J
Code point at index 0: 74

(74 is the Unicode of 'J').
```

## **Example 2: Emoji or Special Symbol**

```
public class Example2 {
    public static void main(String[] args) {
        String str = "Hi ©";

    int codePoint = str.codePointAt(3); // index of '©'

        System.out.println("Character at index 3: " + str.charAt(3));
        System.out.println("Code point at index 3: " + codePoint);
    }
}
```

## **Output:**

Character at index 3: ♥
Code point at index 3: 128522

## **Example 3: Iterating through Characters**

```
public class Example3 {
    public static void main(String[] args) {
        String text = "AB;;

    for (int i = 0; i < text.length(); i++) {
        int cp = text.codePointAt(i);
        System.out.println("Index " + i + " -> " + cp);
     }
    }
}
```

#### **Output:**

```
Index 0 -> 65 (A)
Index 1 -> 66 (B)
Index 2 -> 128522 (©)
```

## Summary Table

Feature Details

**Takes (Argument)** int index (position of character, 0-based)

**Returns** int (Unicode code point of character)

**Use When** Handling Unicode/Emoji/Symbols, or when integer representation

is required

Difference from charAt() → returns char, codePointAt() → returns Unicode

charAt() int

Do you want me to also **compare charAt() and codePointAt() side by side with examples** so it's crystal clear when to use each?

# int codePointBefore(int index)

Got it \_\_\_\_ Let's break down the int codePointBefore(int index) method of Java's String class in a beginner-friendly, structured way.

## Syntax

int codePointBefore(int index)

## Purpose

- Returns the **Unicode code point (integer value)** of the character **just before** the given index in the string.
- It's useful when you want to **inspect characters in reverse** or process Unicode characters (including surrogate pairs for emojis, symbols, etc.).

# Priority in Problem Solving

- **High priority** in **text processing** involving **Unicode or special characters** (like emojis, mathematical symbols, multi-language strings).
- Low priority in simple problems (use charAt() instead for normal characters).

## Parameters

index → position in the string (0-based).
 The method looks at the character just before this index.

## Return Type

int → Unicode code point of the character before the given index.

## **Examples**

## **Example 1: Basic usage**

```
public class CodePointBeforeExample {
  public static void main(String[] args) {
     String str = "Hello";
     int codePoint = str.codePointBefore(1); // character before index 1 = 'H'
     System.out.println("CodePoint: " + codePoint);
     System.out.println("Character: " + (char) codePoint);
  }
}
```

## **Output**:

CodePoint: 72 Character: H

## **Example 2: With last character**

```
public class CodePointBeforeExample2 {
  public static void main(String[] args) {
     String str = "World";
     int codePoint = str.codePointBefore(str.length());
     // last character before index 5 \rightarrow 'd'
     System.out.println("CodePoint: " + codePoint);
     System.out.println("Character: " + (char) codePoint);
```

```
}
}
```

**Output**:

CodePoint: 100 Character: d

#### Example 3: Unicode / Emoji

```
public class CodePointBeforeEmoji {
    public static void main(String[] args) {
        String str = "Hi : ";

    int codePoint = str.codePointBefore(str.length());
    // emoji before index 3
        System.out.println("CodePoint: " + codePoint);
        System.out.println("Character: " + new String(Character.toChars(codePoint)));
    }
}
```

**Output**:

CodePoint: 128522 Character: 😊

# int codePointCount(int beginIndex, int endIndex)

Sure! Let's go step by step for the Java int codePointCount(int beginIndex, int endIndex) method. I'll make it beginner-friendly and precise.

## 1. Syntax

int codePointCount(int beginIndex, int endIndex)

## 2. Purpose

- This method **returns the number of Unicode code points** in the specified text range of a String.
- A **code point** represents a character in Unicode, which can be a single char or a surrogate pair for characters outside the Basic Multilingual Plane (BMP).

## 3. Priority / When to Use

- Medium priority in problem-solving.
- Use when you need accurate character counts for strings that may contain supplementary Unicode characters (like emojis or rare symbols).
- Regular length() counts chars (16-bit), not full Unicode characters.

#### 4. Parameters

Parameter	Type	Description
beginIndex	int	The starting index (inclusive) of the substring in which to count code points.
endIndex	int	The ending index (exclusive) of the substring in which to count code points.

**Note:** Indexing is **0-based**, like other string methods.

## 5. Return Type

• Returns an int, which is the number of Unicode code points in the specified range.

#### 6. Example with Output

```
public class Main {
   public static void main(String[] args) {
      String str = "Hello " World ";

      // Count code points from index 0 to str.length()
      int totalCodePoints = str.codePointCount(0, str.length());
      System.out.println("Total code points: " + totalCodePoints);

      // Count code points in substring "Hello "
      int partCodePoints = str.codePointCount(0, 7); // 7 chars include 1 emoji
      System.out.println("Code points in first 7 chars: " + partCodePoints);
    }
}
```

#### **Output:**

Total code points: 15

Code points in first 7 chars: 6

#### **Explanation:**

- "Hello "World "has 17 chars (length() = 17), but some emojis are surrogate pairs (count as 1 code point each).
- codePointCount() correctly counts **Unicode characters**, not just chars.

If you want, I can also make a **tiny chart comparing length() vs codePointCount()** so you can easily see when to use which in problems.

Do you want me to make that chart?

# int offsetByCodePoints(int index, int codePointOffset)

Sure! Let's break down the Java int offsetByCodePoints(int index, int codePointOffset) method in a clear, beginner-friendly way.

## 1. Syntax

int offsetByCodePoints(int index, int codePointOffset)

#### 2. Purpose

- This method is used to find the **index in a string that is codePointOffset code points away from a given index**.
- A code point is the numeric value of a character in Unicode. It is important when handling characters outside the Basic Multilingual Plane (BMP), such as emoji or certain Asian scripts.

## 3. Priority to Use in Problem Solving

- Use it when you need to accurately navigate a string by Unicode code points, not just by char indices.
- Especially useful in **strings containing surrogate pairs** (characters that take 2 char slots in Java).
- Not very common for basic programming problems but critical for Unicode-safe string processing.

#### 4. Parameters

**Parameter** 

**Description** 

```
index The starting index in the string (0-based).

codePointOff The number of code points to move forward (positive) or backward (negative).
```

## 5. Return Type

 Returns an int, which is the index of the character in the string after moving codePointOffset code points from index.

#### 6. Core Example

```
public class Main {
   public static void main(String[] args) {
      String str = "Hello World"; // is is a 2-char Unicode character
   int index = 5; // starting at
      // Move 1 code point forward
   int newIndex = str.offsetByCodePoints(index, 1);

      // Move 2 code points backward
   int prevIndex = str.offsetByCodePoints(index, -2);

      System.out.println("Original index: " + index);
      System.out.println("Index after moving 1 code point forward: " + newIndex);
      System.out.println("Index after moving 2 code points backward: " + prevIndex);
    }
}
```

## 7. Output

```
Original index: 5
Index after moving 1 code point forward: 7
Index after moving 2 code points backward: 3
```

#### **Explanation:**

- at index 5 uses 2 char slots, so moving 1 code point forward lands at index 7.
- Moving 2 code points backward from index 5 lands at index 3 (1 in "Hello").

If you want, I can also make a **diagram showing exactly how offsetByCodePoints moves through surrogate pairs**—it makes understanding this method much easier.

Do you want me to do that?

# boolean equals(Object another)

Here's a detailed breakdown of the Java method you asked about: boolean equals(Object another) from the String class. I'll cover it in a beginner-friendly way.

#### 1. Syntax

public boolean equals(Object another)

#### 2. Purpose

- The equals() method is used to **compare the content of two strings** (or objects, in general).
- It checks if the values inside the objects are equal, not their memory addresses.
- For strings, == checks references, but equals() checks actual content.

## 3. Priority in Problem Solving

- **High priority** when comparing strings for **equality of content** in algorithms, such as:
  - Checking user input

- Comparing elements in arrays or lists
- Solving string comparison problems in coding challenges
- **Tip:** Always use equals() instead of == for string content comparison.

#### 4. Parameters

Object another

- It takes one argument of type Object.
- This allows it to compare a string with any object.
- If the object is not a string, it will return false.

## 5. Return Type

boolean

- Returns true if the content is **exactly the same**
- Returns false otherwise

## 6. Core Examples

```
public class EqualsExample {
  public static void main(String[] args) {
    String str1 = "Hello";
    String str2 = "Hello";
    String str3 = "hello";
    String str4 = new String("Hello");

// Comparing string contents
  boolean result1 = str1.equals(str2); // true
```

```
boolean result2 = str1.equals(str3); // false (case-sensitive)
boolean result3 = str1.equals(str4); // true (different object, same content)
boolean result4 = str1.equals(null); // false

// Printing results
System.out.println("str1 equals str2? " + result1);
System.out.println("str1 equals str3? " + result2);
System.out.println("str1 equals str4? " + result3);
System.out.println("str1 equals null? " + result4);
}
}
```

#### 7. Output

str1 equals str2? true str1 equals str3? false str1 equals str4? true str1 equals null? false

## Key Notes:

- equals() is case-sensitive.
- If you need case-insensitive comparison, use equalsIgnoreCase().

If you want, I can also make a **small table showing == vs equals()** with examples so you can see when to use each in problem solving. It makes understanding string comparison way easier.

Do you want me to make that?

# boolean equalsIgnoreCase(String another)

Here's a **complete beginner-friendly breakdown** of the Java method you asked about: boolean equalsIgnoreCase(String another)

## 1 Syntax

string1.equalsIgnoreCase(string2)

- string1 → the original string.
- string2 → the string to compare with string1 (case-insensitive).

## 2 Purpose

- Compares two strings ignoring case differences.
- Returns **true** if the strings are equal ignoring case, otherwise **false**.

**Example use case:** Checking user input without worrying about uppercase/lowercase.

## 3 Priority in Problem Solving

- **Wigh priority** when string comparison should **ignore letter case**, e.g., usernames, commands, yes/no inputs.
- Not useful if **case-sensitive** comparison is required.

## 4 Core Example

```
public class Main {
  public static void main(String[] args) {
    String str1 = "HelloWorld";
    String str2 = "helloworld";
    String str3 = "Hello";

    // Comparing str1 and str2
    boolean result1 = str1.equalsIgnoreCase(str2); // true
    System.out.println("str1 equalsIgnoreCase str2: " + result1);

    // Comparing str1 and str3
```

```
boolean result2 = str1.equalsIgnoreCase(str3); // false
    System.out.println("str1 equalsIgnoreCase str3: " + result2);
}
```

#### 5 What it Takes

• Parameter: a single String (cannot be null; if null, returns false).

## 6 Return Type

boolean → true if strings match ignoring case, otherwise false.

## **7** Output of Example

str1 equalsIgnoreCase str2: true str1 equalsIgnoreCase str3: false

If you want, I can make a **mini cheat sheet for all common Java String comparison methods** like equals(), equalsIgnoreCase(), compareTo(), startsWith(), and endsWith() so you can quickly use them in problem-solving.

Do you want me to do that?

# int compareTo(String another)

Sure! Let's break down the Java compareTo(String another) method in a **clear and structured way** for problem-solving.

## 1. Syntax

int result = str1.compareTo(str2);

## 2. Purpose

- The compareTo method compares two strings lexicographically (dictionary order).
- It determines the order of strings for sorting, searching, or conditional logic.

## 3. Priority in Problem Solving

- **High priority** for problems involving:
  - Sorting strings
  - Checking string order
  - Implementing custom string comparison logic
- Not used if you only need equality (equals() is enough).

## 4. Method Signature

public int compareTo(String anotherString)

- Parameter:
  - anotherString → The string you want to compare with the current string.
- Return type: int
  - Negative → Current string is lexicographically less than anotherString.
  - Zero → Strings are equal.
  - o Positive → Current string is lexicographically **greater than** another String.

#### 5. Core Examples

```
public class CompareToExample {
  public static void main(String[] args) {
     String str1 = "apple";
     String str2 = "banana";
     String str3 = "apple";
     // Comparing str1 and str2
     int result1 = str1.compareTo(str2);
     System.out.println("apple vs banana: " + result1); // negative
     // Comparing str2 and str1
     int result2 = str2.compareTo(str1);
     System.out.println("banana vs apple: " + result2); // positive
     // Comparing str1 and str3
     int result3 = str1.compareTo(str3);
     System.out.println("apple vs apple: " + result3); // 0
  }
}
```

## 6. Output Explanation

- apple vs banana: Negative value (because "apple" comes before "banana").
- banana vs apple: Positive value (because "banana" comes after "apple").
- apple vs apple: 0 (strings are equal).

If you want, I can also make a **small table of real compareTo outputs for various strings** so it's easy to memorize for problem-solving.

Do you want me to do that?

# int compareToIgnoreCase(String another)

Sure! Let's break down the Java compareToIgnoreCase(String another) method in a clear, beginner-friendly way.

Method: int compareToIgnoreCase(String another)

## 1. Syntax:

int result = str1.compareTolgnoreCase(str2);

## 2. Purpose:

- Compares two strings lexicographically, ignoring case differences.
- Useful when you want to **sort strings or compare them** without worrying about uppercase/lowercase.

## 3. Priority to use in problem-solving:

- **W** High priority when you need case-insensitive string comparison.
- For example:
  - Sorting a list of names ignoring case.
  - Checking if two strings are equal in a case-insensitive way.
- Use equalsIgnoreCase() if you only want equality check; use compareToIgnoreCase() if you care about order too.

## 4. Parameters (what it takes):

• String another → the string you want to compare with the current string.

#### 5. Return type:

- int
- Meaning of returned value:
  - $\circ$  0  $\rightarrow$  strings are **equal** (ignoring case)
  - $\circ$   $< \theta \rightarrow$  current string is **lexicographically less** than the argument
  - $\circ$  >0  $\rightarrow$  current string is **lexicographically greater** than the argument

## 6. Core Examples:

```
public class CompareTolgnoreCaseExample {
  public static void main(String[] args) {
    String str1 = "Hello";
    String str2 = "hello";
    String str3 = "world";

    // Example 1: Comparing equal strings (ignoring case)
    int result1 = str1.compareTolgnoreCase(str2);
    System.out.println("Result1: " + result1); // Output: 0

    // Example 2: Comparing str1 < str3
    int result2 = str1.compareTolgnoreCase(str3);
    System.out.println("Result2: " + result2); // Output: negative number

    // Example 3: Comparing str3 > str1
    int result3 = str3.compareTolgnoreCase(str1);
    System.out.println("Result3: " + result3); // Output: positive number
}
```

## 7. Output Explanation:

Result1: 0 // "Hello" equals "hello" ignoring case

Result2: -15 // "Hello" is less than "world"
Result3: 15 // "world" is greater than "Hello"

• The exact numbers for <0 or >0 depend on the **Unicode difference of the first** differing character.

If you want, I can also make a **tiny chart showing exactly what values compareToIgnoreCase returns for some string pairs**, which is super useful for quick problem-solving.

Do you want me to do that?

# boolean contentEquals(CharSequence cs)

Here's a **beginner-friendly and detailed explanation** of the Java boolean contentEquals(CharSequence cs) method:

#### 1. Syntax

boolean result = string.contentEquals(CharSequence cs);

## 2. Purpose

- The contentEquals() method compares the content of a String with a given CharSequence (like StringBuilder, StringBuffer, or another String).
- Returns true if both have the **exact same sequence of characters**, otherwise false.
- Useful when you need **exact content comparison**, not reference comparison.

#### 3. Priority to use in problem solving

- **Medium priority** in competitive programming or problem solving.
- Mostly used when:
  - You have a StringBuilder or StringBuffer and want to compare it with a String.
  - You need a character-by-character comparison instead of ==.

#### 4. Parameters it takes

• CharSequence cs → any object that implements CharSequence interface (e.g., String, StringBuilder, StringBuffer).

## 5. Return type

• **boolean** → true if content matches, false otherwise.

#### 6. Core Examples

#### **Example 1: Comparing String with another String**

```
public class Main {
   public static void main(String[] args) {
      String str1 = "Hello";
      String str2 = "Hello";

   boolean result = str1.contentEquals(str2);
      System.out.println(result); // Output: true
   }
}
```

#### **Example 2: Comparing String with StringBuilder**

```
public class Main {
  public static void main(String[] args) {
```

```
String str = "Hello";
StringBuilder sb = new StringBuilder("Hello");

boolean result = str.contentEquals(sb);
System.out.println(result); // Output: true
}
```

#### **Example 3: Comparing String with different content**

```
public class Main {
   public static void main(String[] args) {
      String str = "Hello";
      StringBuffer sb = new StringBuffer("World");

      boolean result = str.contentEquals(sb);
      System.out.println(result); // Output: false
   }
}
```

## Summary:

- contentEquals is for content comparison between String and any CharSequence.
- Returns **true/false** based on exact match.
- Preferred when working with StringBuilder/StringBuffer in problems.

If you want, I can also make a **single Java snippet showing all variations** of contentEquals in one place with their printed results. It's super handy for quick reference.

Do you want me to do that?

# boolean contentEquals(StringBuffer sb)

Sure! Let's break down the contentEquals(StringBuffer sb) method in Java in a beginner-friendly, detailed way.

## 1 Syntax

boolean contentEquals(StringBuffer sb)

## 2 Purpose

- The method compares the content of a String with the content of a StringBuffer.
- It returns true if both contain **exactly the same sequence of characters**, otherwise false.
- Note: It does not compare the memory references or objects, only the characters.

## 3 Priority in Problem Solving

- Moderate priority: Useful when you need to compare a String with a StringBuffer in problems where immutable String vs mutable StringBuffer comparison is required.
- Not used very frequently in standard algorithm problems but can be helpful in string manipulation tasks.

## 4 Parameters

• StringBuffer sb → the StringBuffer whose content you want to compare with the current String.

## **5** Return Type

• **boolean** → returns true if the contents are identical, false otherwise.

## **6** Core Example

```
public class Main {
  public static void main(String[] args) {
    String str = "Hello World";
    StringBuffer sb1 = new StringBuffer("Hello World");
    StringBuffer sb2 = new StringBuffer("Hello Java");

    // Comparing str with sb1
    boolean result1 = str.contentEquals(sb1);
    System.out.println("str vs sb1: " + result1); // true

    // Comparing str with sb2
    boolean result2 = str.contentEquals(sb2);
    System.out.println("str vs sb2: " + result2); // false
  }
}
```

## **7** Output

str vs sb1: true str vs sb2: false

## **Explanation:**

- $str.contentEquals(sb1) \rightarrow true because "Hello World" == "Hello World"$
- str.contentEquals(sb2) → false because "Hello World" != "Hello Java"

If you want, I can make a **small table of all String comparison methods** including contentEquals, equals. equalsIgnoreCase, compareTo, showing when to use each in problem-solving. It will be super handy for coding interviews.

Do you want me to do that?

# boolean matches(String regex)

Sure! Let's break down the boolean matches(String regex) method in Java **step by step**. I'll make it beginner-friendly and include examples with output.

## 1 Syntax

boolean result = str.matches(String regex);

## 2 Purpose

The matches() method checks if the **entire string** matches a **regular expression (regex)** pattern.

- If the string fully matches the regex → returns true
- Otherwise → returns false

**Important:** It matches the **whole string**, not part of it.

## Priority to Use in Problem Solving

- Useful in input validation, e.g., checking if a string is a valid email, phone number, or number.
- Great for pattern matching problems.
- Not efficient for partial matching (use Pattern and Matcher for complex scenarios).

## 4 Parameters

• String regex → the regular expression to match against the string.

## 5 Return Type

boolean → true if string matches regex, false otherwise.

## **6** Core Examples

```
public class MatchesExample {
  public static void main(String[] args) {
     String str1 = "12345";
     String str2 = "abc123";
     String str3 = "hello";
     // Example 1: digits only
     boolean res1 = str1.matches("\\d+"); // \\d+ means one or more digits
     System.out.println("str1 matches digits? " + res1); // true
     // Example 2: letters only
     boolean res2 = str2.matches("[a-zA-Z]+"); // letters only
     System.out.println("str2 matches letters? " + res2); // false
     // Example 3: exact word
     boolean res3 = str3.matches("hello"); // exact match
     System.out.println("str3 matches 'hello'? " + res3); // true
     // Example 4: letters and numbers
     boolean res4 = str2.matches("[a-zA-Z0-9]+"); // letters or digits
     System.out.println("str2 matches letters/digits? " + res4); // true
  }
}
```

## Sample Output

```
str1 matches digits? true
str2 matches letters? false
str3 matches 'hello'? true
str2 matches letters/digits? true
```

#### **✓** Summary:

- Use matches() for full string regex matching.
- Returns a boolean.
- Perfect for validation problems in coding challenges.

If you want, I can make a **small table of common regex patterns with matches()** that is super handy for solving coding problems quickly.

Do you want me to do that?

# boolean contains(CharSequence seq)

Sure! Let's break down the Java boolean contains (CharSequence seq) method in a beginner-friendly, detailed way.

#### 1. Syntax

boolean result = string.contains(CharSequence seq);

#### 2. Purpose

The contains() method checks whether a given sequence of characters (CharSequence) is present inside the string.

- If the sequence exists → returns true
- If it doesn't exist → returns false

**Key point:** It is case-sensitive.

#### 3. Priority in Problem Solving

#### **✓ High Priority** in string search problems:

- Checking substrings
- Validating input
- Searching patterns in strings
- Pre-check before further operations like replace() or index0f()

# 4. Parameters (What it takes)

CharSequence seq

- CharSequence can be:
  - o String
  - StringBuilder
  - StringBuffer

#### Example:

```
String str = "Hello World";
CharSequence seq1 = "World";
CharSequence seq2 = new StringBuilder("Hello");
```

# 5. Return Type

boolean

- true  $\rightarrow$  if the sequence is found
- false  $\rightarrow$  if the sequence is not found

#### 6. Core Examples

```
public class ContainsExample {
  public static void main(String[] args) {
     String str = "Hello World";
     // Example 1: Using String
     boolean result1 = str.contains("World"); // true
     // Example 2: Using StringBuilder
     StringBuilder sb = new StringBuilder("Hello");
     boolean result2 = str.contains(sb);
                                            // true
     // Example 3: Sequence not present
     boolean result3 = str.contains("Java"); // false
     // Printing the results
     System.out.println("Contains 'World'?: " + result1);
     System.out.println("Contains 'Hello' via StringBuilder?: " + result2);
     System.out.println("Contains 'Java'?: " + result3);
  }
}
```

#### **Output:**

Contains 'World'?: true

Contains 'Hello' via StringBuilder?: true

Contains 'Java'?: false

If you want, I can make a **small table of all important Java String methods** like contains() with their **priority and examples** so you can use it as a quick reference for problem solving.

Do you want me to make that table?

# boolean startsWith(String prefix)

Here's a **complete**, **beginner-friendly explanation** for the Java startsWith(String prefix) method:

#### 1. Syntax

boolean startsWith(String prefix)

Or the overloaded version with an index:

boolean startsWith(String prefix, int toffset)

#### 2. Purpose

- Checks if the string starts with the specified prefix.
- Useful in searching, filtering, and validating strings.

#### 3. Priority in Problem Solving

- **Medium priority**: Often used in string manipulation problems, such as checking file extensions, commands, or URL patterns.
- Not as commonly used as equals() or contains(), but very handy in conditional checks.

#### 4. Parameters

- prefix: The substring you want to check at the start of the string.
- toffset (optional): The index in the string where checking should start.

#### 5. Return Type

#### • Returns boolean:

- $\circ$  true  $\rightarrow$  if the string starts with the given prefix.
- $\circ$  false  $\rightarrow$  otherwise.

#### 6. Core Examples

```
public class StartsWithExample {
   public static void main(String[] args) {
      String str1 = "HelloWorld";
      String str2 = "JavaProgramming";

      // Basic usage
      boolean result1 = str1.startsWith("Hello"); // true
      boolean result2 = str1.startsWith("World"); // false

      // Using offset
      boolean result3 = str2.startsWith("Programming", 4); // true (starts checking at index 4)

      // Printing the results
      System.out.println(result1); // true
      System.out.println(result2); // false
      System.out.println(result3); // true
   }
}
```

## **Output**

true false true

If you want, I can also make a **small table showing all important string methods with their priority and use cases** like this for fast reference in problem solving. It's a great cheat sheet for coding interviews.

Do you want me to make that?

# boolean startsWith(String prefix, int toffset)

Sure! Let's break it down **step by step** for the Java startsWith(String prefix, int toffset) method:

#### 1. Syntax

boolean startsWith(String prefix, int toffset)

#### 2. Purpose

- Checks if the substring of this string starting at the specified index (toffset) begins with the specified prefix (prefix).
- Useful when you want to verify a pattern or prefix starting at a particular position in a string.

# 3. Priority in Problem Solving

- Medium Priority:
  - Frequently used in string pattern matching, parsing, and validation problems.
  - Less common than the simpler startsWith(String prefix) which always checks from index 0.

#### 4. Parameters

Parameter	Description
prefix	The prefix string you want to check for.
toffset	The starting index in the main string where the check should begin.

#### 5. Return Type

boolean

- Returns true if the substring starting at toffset starts with prefix.
- Returns false otherwise.

#### 6. Example Usage

```
public class StartsWithExample {
   public static void main(String[] args) {
      String str = "HelloWorld";

      // Check if substring starting at index 5 starts with "World"
      boolean result1 = str.startsWith("World", 5);
      System.out.println(result1); // true

      // Check if substring starting at index 0 starts with "World"
      boolean result2 = str.startsWith("World", 0);
      System.out.println(result2); // false

      // Check if substring starting at index 3 starts with "loW"
      boolean result3 = str.startsWith("loW", 3);
      System.out.println(result3); // true
   }
}
```

#### 7. Output

true false true



- Indexing is **0-based**.
- If toffset is out of bounds (negative or greater than string length), it throws StringIndexOutOfBoundsException.

If you want, I can also make a **small table showing all Java startsWith variations with examples** to make it very easy to memorize for problem-solving.

Do you want me to do that?

# boolean endsWith(String suffix)

Sure! Let's go **step by step** for the Java String method:

## Method: boolean endsWith(String suffix)

# 1 Syntax

stringVariable.endsWith(suffix);

- **stringVariable** → the string you are testing.
- **suffix** → the string you want to check if stringVariable ends with.

## 2 Purpose

- To check if a string ends with a specified suffix.
- Returns true if the string ends with that suffix, otherwise false.

#### **3** Priority in Problem Solving

- Medium priority
- Useful when solving problems related to:
  - File extensions (e.g., .txt, .java)
  - o URL endings
  - Pattern matching at the end of strings

#### 4 Arguments it Takes

• A single argument: String suffix

# 5 Return Type

• boolean → true or false

# 6 Core Examples

```
public class EndsWithExample {
  public static void main(String[] args) {
    String str1 = "HelloWorld";
    String str2 = "Welcome.java";

    // Example 1
    boolean result1 = str1.endsWith("World"); // true
    System.out.println("Does str1 end with 'World'? " + result1);

    // Example 2
    boolean result2 = str2.endsWith(".java"); // true
    System.out.println("Does str2 end with '.java'? " + result2);

    // Example 3
```

```
boolean result3 = str1.endsWith("Hello"); // false
    System.out.println("Does str1 end with 'Hello'? " + result3);
}
```

#### **7** Sample Output

Does str1 end with 'World'? true Does str2 end with '.java'? true Does str1 end with 'Hello'? false

If you want, I can also make a **small table with all important String methods** like endsWith, startsWith, contains, etc., showing their syntax, return type, and usage, which is super handy for interviews and problem solving.

Do you want me to do that?

# int indexOf(int ch)

Sure! Let's break it down **step by step** for the Java method: int indexOf(int ch) from the String class.

#### 1. Syntax

int indexOf(int ch)

#### 2. Purpose

- This method **returns the index of the first occurrence** of the specified character (ch) in the string.
- If the character is **not found**, it returns -1.
- It's commonly used when you need to **search for a character's position** in a string.

#### 3. Priority in Problem Solving

- High priority for string problems where:
  - You need to find positions of characters.
  - You want to split strings or validate character positions.
  - You want to **check if a character exists** before performing operations.

#### 4. Parameters

- ch → the character to search for.
  - Can be provided as:
    - Character literal: 'a'
    - ASCII/Unicode value: 97 (for 'a')

#### 5. Return Type

- Returns an int → index of first occurrence of the character.
- Returns -1 if the character is **not present** in the string.

#### 6. Examples

```
public class IndexOfExample {
   public static void main(String[] args) {
      String str = "Hello World";

      // Example 1: Using character literal
   int index1 = str.indexOf('o'); // first 'o' occurs at index 4
      System.out.println("Index of 'o': " + index1);
```

```
// Example 2: Using ASCII value
int index2 = str.indexOf(87); // ASCII 87 = 'W'
   System.out.println("Index of 'W': " + index2);

// Example 3: Character not present
int index3 = str.indexOf('z'); // 'z' not present
   System.out.println("Index of 'z': " + index3);
}
```

#### 7. Output

Index of 'o': 4 Index of 'W': 6 Index of 'z': -1

#### Summary:

- Use indexOf(int ch) to quickly find the first occurrence of a character.
- Returns **position** if found, **-1** if not.
- Very useful for string parsing, validations, and searching.

If you want, I can also create a **small table of all indexOf variations** in Java and when to use them, which is super handy for problem solving.

Do you want me to do that?

# int indexOf(int ch, int fromIndex)

Absolutely! Let's break down the Java String method you mentioned: int index0f(int ch, int fromIndex) in a beginner-friendly, structured way.



#### 2 Purpose

- This method returns the index of the first occurrence of the specified character (ch) in the string starting from the given index (fromIndex).
- If the character is **not found**, it returns -1.

# 3 Priority in Problem Solving

- High Priority for problems where you need to find positions of characters in strings.
- Useful in:
  - Searching for characters after a certain position.
  - Parsing or extracting substrings.
  - o Implementing algorithms like pattern matching.

## 4 Parameters

Parameter	Type	Description
ch	int	Character (or ASCII code) to search for
fromInde x	int	Index to start the search from (0-based)

Note: int ch can also be a character like 'a', which is automatically converted to its ASCII int value.

#### **5** Return Type

- int → index of the **first occurrence** of ch after fromIndex.
- Returns -1 if not found.

#### 6 Core Examples

```
public class IndexOfExample {
    public static void main(String[] args) {
        String str = "hello world";

        // Example 1: Find 'o' starting from index 0
        int idx1 = str.indexOf('o', 0);
        System.out.println("Index of 'o' from 0: " + idx1); // Output: 4

        // Example 2: Find 'o' starting from index 5
        int idx2 = str.indexOf('o', 5);
        System.out.println("Index of 'o' from 5: " + idx2); // Output: 7

        // Example 3: Character not found
        int idx3 = str.indexOf('z', 0);
        System.out.println("Index of 'z': " + idx3); // Output: -1
    }
}
```

## **7** Output

```
Index of 'o' from 0: 4
Index of 'o' from 5: 7
Index of 'z': -1
```

## Summary:

- indexOf(int ch, int fromIndex) → finds character position after a given index.
- Returns -1 if not found.

Very useful in string searching, parsing, and problem solving.

If you want, I can also make a **small table showing all indexOf variants with examples**, so you can use them like a cheat sheet for problem solving.

Do you want me to do that?

# int indexOf(String str)

Sure! Let's break down the **int indexOf(String str)** method in Java in a beginner-friendly, structured way.

# 1 Syntax

int index = str.indexOf(substring);

- str → the main string you are searching in.
- substring → the string whose first occurrence index you want to find.
- index → stores the index of the first occurrence.

#### 2 Purpose

- To find the index of the first occurrence of a specified substring in a string.
- Returns -1 if the substring is **not found**.

#### Use case:

Whenever you need to check the **position of a word/character** in a string for string manipulation, parsing, or validation.

# Priority in Problem Solving

- **High priority** in string manipulation problems.
- Useful in:
  - Searching for a substring
  - Splitting strings
  - Validating presence of a word
  - String parsing in algorithmic problems

#### 4 Parameters it takes

- **Single parameter:** String str → the substring to search.
- Example: "Hello World".indexOf("World") → looks for "World" in "Hello World".

# **5** Return Type

- Returns int → index of the first occurrence (0-based).
- Returns -1 if not found.

#### **6** Core Examples

```
public class IndexOfExample {
  public static void main(String[] args) {
    String text = "Hello World";

    // Example 1: substring exists
    int index1 = text.indexOf("World"); // returns 6
    System.out.println("Index of 'World': " + index1);

    // Example 2: substring does not exist
```

```
int index2 = text.indexOf("Java"); // returns -1
    System.out.println("Index of 'Java': " + index2);

// Example 3: first occurrence of a character
    int index3 = text.indexOf("o"); // returns 4 (first 'o')
    System.out.println("Index of first 'o': " + index3);
}
```

## **7** Output

Index of 'World': 6 Index of 'Java': -1 Index of first 'o': 4

If you want, I can also give a **super handy tip** on how to use index0f in **more advanced problems** like finding all occurrences of a substring efficiently.

Do you want me to show that?

# int indexOf(String str, int fromIndex)

Sure! Let's break down the int indexOf(String str, int fromIndex) method in Java in a clear and structured way.

## 1. Syntax

int indexOf(String str, int fromIndex)

#### 2. Purpose

- Finds the **first occurrence** of the specified substring (str) **starting from the given index** (fromIndex) in the calling string.
- If the substring is **not found**, it returns -1.

#### 3. Priority in Problem Solving

- **High priority** when you need to:
  - Search for substrings after a certain position.
  - Avoid scanning the string from the beginning multiple times.
  - Useful in string parsing, validation, or pattern checking problems.

#### 4. Core Example

```
public class Main {
   public static void main(String[] args) {
      String text = "Hello world! Welcome to Java world.";

      // Search for "world" starting from index 0
      int firstIndex = text.indexOf("world", 0);

      // Search for "world" starting from index 12
      int secondIndex = text.indexOf("world", 12);

      // Search for "Python" starting from index 0
      int notFound = text.indexOf("Python", 0);

      // Print results
      System.out.println("First occurrence of 'world' from index 0: " + firstIndex);
      System.out.println("Occurrence of 'world' from index 12: " + secondIndex);
      System.out.println("Occurrence of 'Python' from index 0: " + notFound);
   }
}
```

#### 5. What It Takes

- str: The substring to search for.
- fromIndex: The starting index from where the search begins.

#### 6. Return Type

- int: The index of the first occurrence of the substring after fromIndex.
- Returns -1 if the substring is not found.

#### 7. Sample Output

First occurrence of 'world' from index 0: 6 Occurrence of 'world' from index 12: 29 Occurrence of 'Python' from index 0: -1

#### Summary Tip:

• Use indexOf(str, fromIndex) when you need **controlled searching** in strings from a specific index rather than from the beginning.

If you want, I can also make a **small visual diagram** showing how indexOf(str, fromIndex) searches the string—it makes it super easy to remember for problem-solving.

Do you want me to do that?

# int lastIndexOf(int ch)

Sure! Let's break down the Java method int lastIndexOf(int ch) in a clear and structured way:

# 1. Syntax

int lastIndexOf(int ch)

or if you are using it with a String object:

```
int index = str.lastIndexOf(ch);
```

#### 2. Purpose

- The method searches for the **last occurrence** of the specified character ch in the string.
- Returns the **index** of that character (0-based).
- If the character is **not found**, it returns -1.

#### 3. Priority to Use in Problem Solving

- **Medium priority**: Useful when you need to find the **position of the last occurrence** of a character in a string.
- Common scenarios:
  - Parsing file paths (/ or \ separator)
  - Finding last occurrence of a delimiter
  - String manipulation where end positions matter

## 4. Core Examples

```
public class LastIndexOfExample {
  public static void main(String[] args) {
    String str = "hello world";

    // Finding last index of 'I'
    int index1 = str.lastIndexOf('I');
    System.out.println("Last index of 'I': " + index1);

    // Finding last index of 'o'
    int index2 = str.lastIndexOf('o');
    System.out.println("Last index of 'o': " + index2);
```

```
// Character not in string
int index3 = str.lastIndexOf('z');
    System.out.println("Last index of 'z': " + index3);
}
```

#### 5. What It Takes

• Parameter: A single character (as int or char) to search for.

#### Example:

```
str.lastIndexOf('a'); // 'a' as char
str.lastIndexOf(97); // 97 is ASCII of 'a'
```

•

## 6. Return Type

- int: The index of the last occurrence of the character.
- Returns -1 if the character is not found.

#### 7. Example Output

For the code above, the output will be:

Last index of 'I': 9 Last index of 'o': 7 Last index of 'z': -1

If you want, I can also make a **quick visual diagram** showing how lastIndexOf() searches from the end of a string—it helps in remembering it for problem-solving.

Do you want me to make that diagram?

# int lastIndexOf(int ch, int fromIndex)

Sure! Let's break down everything about the lastIndexOf(int ch, int fromIndex) method in Java. I'll give a detailed explanation with examples and output.

#### 1. Syntax

int lastIndexOf(int ch, int fromIndex)

#### 2. Purpose

- Finds the last occurrence of the character ch starting from the specified index fromIndex and moving backward.
- Useful when you want to search for a character in a string from a specific position toward the beginning.

#### 3. Priority in Problem Solving

- Medium priority.
- Frequently used in string manipulation problems where:
  - You need the last occurrence of a character before a certain position.
  - o You are parsing strings in reverse order.
- Example use cases: finding last delimiter, checking for repeated characters, substring extraction from the end.

#### 4. Parameters

**Parameter** 

**Description** 

```
int ch The character (Unicode value) you are searching for.

int The index from where to start searching backward (0-based).
```

#### 5. Return Type

- Returns int:
  - The index of the last occurrence of ch before or at fromIndex.
  - o Returns -1 if the character is **not found**.

#### 6. Core Examples

```
public class LastIndexOfExample {
   public static void main(String[] args) {
        String str = "hello world";

        // Find last index of 'o' starting from index 7
        int index1 = str.lastIndexOf('o', 7);
        System.out.println("Last index of 'o' from index 7: " + index1);

        // Find last index of 'l' starting from index 3
        int index2 = str.lastIndexOf('l', 3);
        System.out.println("Last index of 'l' from index 3: " + index2);

        // Character not present
        int index3 = str.lastIndexOf('z', 5);
        System.out.println("Last index of 'z' from index 5: " + index3);
    }
}
```

## 7. Output

```
Last index of 'o' from index 7: 4
Last index of 'l' from index 3: 3
Last index of 'z' from index 5: -1
```

#### **Summary**

- lastIndexOf(int ch, int fromIndex) → searches backward from a given index.
- Returns index of last occurrence or -1.
- Useful for string parsing and reverse search problems.

If you want, I can also make a **table of all lastIndexOf variations** with simple examples for each. It'll be very handy for problem solving. Do you want me to do that?

# int lastIndexOf(String str)

Sure! Let's go step by step for the Java lastIndexOf(String str) method. I'll cover syntax, purpose, priority, examples, return type, and printing the value.

#### 1. Syntax

int lastIndexOf(String str)

- $str \rightarrow the$  substring whose **last occurrence** you want to find in the string.
- Returns the **index** of the last occurrence of the substring.

Optional variant with a starting position:

int lastIndexOf(String str, int fromIndex)

• Searches backward from fromIndex.

#### 2. Purpose

- To find the **last occurrence** of a substring in a string.
- Useful when you need to locate where a substring appears last, not first.
- Common in parsing strings, checking patterns, and substring manipulations.

#### 3. Priority in Problem Solving

- Medium priority.
- Used when string analysis requires the last position of a substring.
- Not always needed in basic problems, but essential in parsing, reverse search, and some algorithmic string manipulations.

#### 4. Core Examples

```
public class LastIndexOfExample {
  public static void main(String[] args) {
    String text = "hello world, hello Java";

    // Find last occurrence of "hello"
    int lastIndex1 = text.lastIndexOf("hello");

    // Find last occurrence of "o"
    int lastIndex2 = text.lastIndexOf("o");

    // Find last occurrence of "hello" before index 10
    int lastIndex3 = text.lastIndexOf("hello", 10);

    // Print results
    System.out.println("Last index of 'hello': " + lastIndex1); // 13
    System.out.println("Last index of 'o': " + lastIndex2); // 16
    System.out.println("Last index of 'hello' before index 10: " + lastIndex3); // 0
}
```

## 5. What It Takes (Parameters)

- A **string** (str) to search for.
- Optionally, an **int index** to start backward search.

## 6. Return Type

- $int \rightarrow the index of the last occurrence of the substring.$
- Returns -1 if the substring is **not found**.

## 7. Printing Its Value

As shown in the example:

System.out.println(text.lastIndexOf("hello")); // Output: 13

• Prints the position of the last occurrence.

## Summary Table

Aspect	Details
Syntax	<pre>int lastIndexOf(String str)</pre>
Purpose	Find last occurrence of substring
Priority	Medium
Parameters	String str, optionally int fromIndex

Return Type int (index or -1)

Example Output

13, 16, 0

If you want, I can make a **small visual diagram** showing how lastIndexOf works with a string, which makes it **super easy to memorize**.

Do you want me to make that diagram?

# int lastIndexOf(String str, int fromIndex)

Sure! Let's break down the Java method you mentioned step by step.

#### Method: int lastIndexOf(String str, int fromIndex)

#### 1. Syntax

int lastIndexOf(String str, int fromIndex)

- **str** → the substring you want to search for.
- **fromIndex** → the index from where the search starts **backward** in the string.

#### 2. Purpose

- Finds the **last occurrence** of the specified substring in a string **starting from a given** index and searching backwards.
- Useful when you need the **rightmost position** of a substring rather than the first occurrence.

#### 3. Priority in Problem Solving

- **Medium to High**: Frequently used in string manipulation problems, such as:
  - Extracting file extensions (lastIndexOf("."))
  - Parsing URLs or paths
  - Reversing searches in strings
- Not always the first method to reach for, but very useful for backward searches.

#### 4. Core Examples

```
public class LastIndexOfExample {
  public static void main(String[] args) {
    String text = "hello world, hello Java";

  // Find last occurrence of "hello" starting backward from index 15
  int index1 = text.lastIndexOf("hello", 15);
    System.out.println("Last occurrence of 'hello' from index 15: " + index1);

  // Find last occurrence of "Java" starting backward from index 10
  int index2 = text.lastIndexOf("Java", 10);
    System.out.println("Last occurrence of 'Java' from index 10: " + index2);

  // Last occurrence of 'o' from index 12
  int index3 = text.lastIndexOf("o", 12);
    System.out.println("Last occurrence of 'o' from index 12: " + index3);
  }
}
```

#### 5. What it Takes (Parameters)

- 1. String  $str \rightarrow Substring$  to find.
- 2. int fromIndex  $\rightarrow$  Index to start searching backwards.

#### 6. Return Type

- Returns an int:
  - The index of the last occurrence of the substring (≤ fromIndex)
  - Returns **-1** if the substring is not found.

#### 7. Example Output

Last occurrence of 'hello' from index 15: 0 Last occurrence of 'Java' from index 10: -1 Last occurrence of 'o' from index 12: 7

If you want, I can make a **quick diagram showing how fromIndex affects the search direction**, which makes it **super easy to remember for problem solving**.

Do you want me to do that?

# String substring(int beginIndex)

Sure! Let's break it down thoroughly for the Java String substring(int beginIndex) method. I'll provide all the details you asked for.

# 1. Syntax

String substring(int beginIndex)

- beginIndex → the starting index (0-based) from where you want to extract the substring.
- Index starts at 0.
- The substring will go from beginIndex to the end of the string.

#### 2. Purpose

- To extract a part of a string starting from a specified index up to the **end of the string**.
- Useful in string manipulation problems, like extracting names, suffixes, or parts of a message.

#### 3. Priority to Use in Problem Solving

- **High** for problems involving:
  - Parsing strings
  - Extracting substrings
  - Slicing data for comparisons
  - Substring-based logic in algorithm questions (like checking suffixes, prefixes, or removing a part of a string)

#### 4. Core Examples

```
Example 1: Basic usage
```

```
public class Main {
   public static void main(String[] args) {
      String str = "HelloWorld";
      String subStr = str.substring(5); // starts from index 5
      System.out.println(subStr);
   }
}
```

#### **Output:**

World

**Explanation:** Index 5 is 'W'. It extracts everything from 'W' to the end.

#### **Example 2: Using with variables**

```
public class Main {
   public static void main(String[] args) {
      String str = "Programming";
      int startIndex = 3;
      String subStr = str.substring(startIndex);
      System.out.println(subStr);
   }
}
```

#### **Output:**

gramming

**Explanation:** Extracts substring starting from index 3.

#### 5. Return Type

String → The method returns a new string that is a substring of the original string.

#### 6. Notes

- Throws IndexOutOfBoundsException if beginIndex < 0 or beginIndex > length of string.
- Cannot pass endIndex in this version (use substring(int beginIndex, int endIndex) for that).

I can also make a **small table with beginIndex vs Output** for clarity.

Do you want me to make that table?

# String substring(int beginIndex, int endIndex)

Sure! Let's break down the **String substring(int beginIndex, int endIndex)** method in Java thoroughly:

#### 1. Syntax

String substring(int beginIndex, int endIndex)

- **beginIndex** → Starting index (inclusive)
- **endIndex** → Ending index (exclusive)

The substring will include characters from beginIndex to endIndex - 1.

#### 2. Purpose

- To extract a portion of a string between two specified indices.
- Commonly used in parsing strings, splitting data, or manipulating strings for problem-solving.

#### 3. Priority to use in problem solving

- **High priority**: Almost every string manipulation problem on platforms like LeetCode, HackerRank, etc., requires substring extraction.
- Useful in:
  - Checking prefixes/suffixes
  - Extracting numbers or words from strings
  - Reversing parts of a string

#### 4. Core examples

```
public class SubstringExample {
   public static void main(String[] args) {
      String text = "HelloWorld";

      // Extract "Hello"
      String part1 = text.substring(0, 5);

      // Extract "World"
      String part2 = text.substring(5, 10);

      // Extract "loWo"
      String part3 = text.substring(3, 7);

      System.out.println("part1 = " + part1); // Hello
      System.out.println("part2 = " + part2); // World
      System.out.println("part3 = " + part3); // loWo
    }
}
```

#### 5. What it takes

- Two integer parameters: beginIndex and endIndex.
- Constraints:

```
beginIndex >= 0endIndex <= length of string</li>
```

beginIndex <= endIndex</li>
 If these are violated, it throws StringIndexOutOfBoundsException.

#### 6. Return Type

- Returns a new String containing the specified substring.
- Original string remains unchanged (immutable nature of strings).

#### 7. Output from above example

part1 = Hello part2 = World part3 = IoWo

If you want, I can also make a **table of all common substring() use cases** with examples and their output—super handy for LeetCode-style string problems.

Do you want me to do that?

# CharSequence subSequence(int beginIndex, int endIndex)

Absolutely! Let's break down the subSequence(int beginIndex, int endIndex) method in Java step by step, covering everything you asked.

# 1 Syntax

CharSequence subSequence(int beginIndex, int endIndex)

## 2 Purpose

- The subSequence() method returns a new sequence of characters that is a sub-part of the original string (or any object implementing CharSequence).
- The characters returned start from beginIndex (inclusive) and go up to endIndex (exclusive).

• Useful when you need a portion of a string for processing or comparison.

#### 3 Priority in Problem Solving

- Medium to high priority in string manipulation problems.
- Use it when:
  - Extracting substrings without converting the object to String.
  - Working with APIs requiring CharSequence.
- Almost equivalent to substring(), but subSequence() returns CharSequence, not String.

#### 4 Core Example

- Input: a String or StringBuilder object.
- Parameters:
  - $\circ$  beginIndex  $\rightarrow$  starting index (inclusive)
  - endIndex → ending index (exclusive)
- Return type: CharSequence

```
public class Main {
  public static void main(String[] args) {
    String str = "HelloWorld";

    // Using subSequence
    CharSequence subSeq = str.subSequence(0, 5);

    // Print the returned value
    System.out.println("SubSequence: " + subSeq); // Output: Hello
}
```

#### 5 Key Points

Feature Details

Method belongs to CharSequence interface (implemented by String,

StringBuilder, etc.)

Parameters int beginIndex, int endIndex

Return Type Char Sequence

Inclusive/Exclusive beginIndex inclusive, endIndex exclusive

Exception Throws IndexOutOfBoundsException if indices are invalid

Difference with Returns CharSequence instead of String

substring()

## **6** Output of Example

SubSequence: Hello

If you want, I can also **show a quick example with StringBuilder and compare subSequence() vs substring()** which often confuses beginners.

Do you want me to do that?

# String concat(String str)

Sure! Let's break down the **String concat(String str)** method in Java step by step in a structured way.

#### 1. Syntax

String newString = originalString.concat(str);

- originalString → the string you want to append to.
- $str \rightarrow the string you want to concatenate (append) to original String.$
- newString → the resulting string after concatenation.

#### 2. Purpose

- The concat() method is used to combine (append) two strings.
- It does not modify the original string because strings in Java are immutable. Instead, it returns a **new string**.

# 3. Priority in Problem Solving

- Moderate priority:
  - Useful in basic string manipulations, like combining names, paths, or text.
  - In competitive programming, using + operator is more common because it is simpler.
  - concat() is slightly more efficient than + in some cases, but mostly both are fine for readability.

# 4. Core Examples

```
public class ConcatExample {
   public static void main(String[] args) {
      // Original string
      String str1 = "Hello";

   // String to append
      String str2 = " World";
```

```
// Using concat method
   String result = str1.concat(str2);

// Print the result
   System.out.println(result);

// Example showing immutability
   System.out.println("Original str1: " + str1);
}
```

#### **Output:**

Hello World Original str1: Hello

# 5. What it Takes (Parameters)

- Takes exactly one parameter:
  - $\circ$  String str  $\rightarrow$  the string to append.
- Throws NullPointerException if str is null.

# 6. Return Type

• Returns **String** → the new concatenated string.

# Summary Table

Feature	Details
Syntax	str1.concat(str2)
Purpose	Combine two strings into one

Parameter(s) String str

Return Type String

Original String

Change

No (immutable)

Usage Priority Moderate (useful for clarity)

Throws NullPointerException if

argument null

If you want, I can also make a **mini chart showing concat() vs + operator vs**StringBuilder.append(), which is super useful for problem-solving efficiency in Java.

Do you want me to make that?

# String replace(char oldChar, char newChar)

Sure! Let's break down the **String replace(char oldChar, char newChar)** method in Java step by step:

# 1. Syntax

String newString = originalString.replace(oldChar, newChar);

- originalString → the string in which you want to replace characters
- oldChar → the character you want to replace
- newChar → the character you want it replaced with

# 2. Purpose

The purpose of this method is to replace all occurrences of a specific character in a string with another character.

# 3. Priority in Problem Solving

- Medium priority: Useful in string manipulation problems, data cleanup, formatting strings, or replacing unwanted characters.
- Not for complex replacements like regex patterns; for that, use replaceAll().

#### 4. Core Examples

#### **Example 1: Replace a single character**

```
String str = "hello";
String replacedStr = str.replace('I', 'p'); // replaces all 'I' with 'p'
System.out.println(replacedStr); // heppo
```

#### **Example 2: No occurrence of oldChar**

```
String str = "hello";
String replacedStr = str.replace('x', 'y'); // 'x' not present
System.out.println(replacedStr); // hello
```

#### **Example 3: Replace all same characters**

```
String str = "aaaaa";
String replacedStr = str.replace('a', 'b');
System.out.println(replacedStr); // bbbbb
```

# 5. Return Type

- The method **returns a new String** with the replaced characters.
- Original string remains **unchanged** because strings are immutable in Java.

#### 6. Printing Its Value

String str = "programming"; String replacedStr = str.replace('g', 'x'); System.out.println("Original: " + str); System.out.println("Replaced: " + replacedStr);

#### **Output:**

Original: programming Replaced: proxxramminx

# Summary Table

**Feature Details** Syntax str.replace(oldChar, newChar) Purpose Replace all occurrences of oldChar with newChar **Priority** Medium (common in string manipulation problems) Return Type String (new string) Original String Not modified (immutable) Core Replace single char, replace multiple chars, no Examples occurrence

If you want, I can also give a **small visual diagram showing how it replaces characters step by step** — it helps in remembering it for problem-solving.

Do you want me to make that diagram?

# String replace(CharSequence target, CharSequence replacement)

Absolutely! Let's break down the String.replace(CharSequence target, CharSequence replacement) method in Java in a structured way for problem-solving.

#### 1. Syntax

String newString = originalString.replace(CharSequence target, CharSequence replacement);

#### 2. Purpose

- Replaces **all occurrences** of a given character sequence (target) in the original string with another character sequence (replacement).
- Works with strings, characters, or any object that implements CharSequence.
- Does **not modify the original string** (Strings are immutable). It returns a new string.

## 3. Priority to Use in Problem Solving

- High Priority for string manipulation problems where:
  - You need to replace substrings.
  - You want a simple and direct solution without loops.
- Often used in problems like:
  - Removing unwanted characters (e.g., whitespace, symbols).
  - Converting formats (e.g., changing separators in CSV).
  - Replacing placeholders in templates.

# 4. Core Examples

```
public class ReplaceExample {
   public static void main(String[] args) {
      String original = "I love Java, Java is fun!";

   // Replace word "Java" with "Python"
   String replaced1 = original.replace("Java", "Python");
```

```
System.out.println(replaced1);

// Output: I love Python, Python is fun!

// Replace single character 'a' with 'o'
String replaced2 = original.replace("a", "o");
System.out.println(replaced2);

// Output: I love Jovo, Jovo is fun!

// Replace substring "love Java" with "like Python"
String replaced3 = original.replace("love Java", "like Python");
System.out.println(replaced3);

// Output: I like Python, Java is fun!
}
```

#### 5. What It Takes

- Parameters:
  - $\circ$  target  $\rightarrow$  CharSequence to be replaced
  - o replacement → CharSequence to replace with
- Return Type:
  - String → a new string with replacements applied

# 6. Printing Its Value

Already shown in examples above using:

```
System.out.println(replaced1);
System.out.println(replaced2);
System.out.println(replaced3);
```

Feature Details

Syntax string.replace(target, replacement)

Purpose Replace all occurrences of target

Parameter Types CharSequence target, CharSequence

replacement

Return Type String (new string)

Priority in Problems High for string manipulation

Example Use Remove characters, change substrings,

Cases formatting

If you want, I can also create a **small visual cheat sheet of all String.replace()** variations with examples for quick reference in coding interviews.

Do you want me to do that?

# String replaceAll(String regex, String replacement)

Sure! Let's break down the String replaceAll(String regex, String replacement) method in Java in detail, exactly as you requested.

#### 1. Syntax

String newString = originalString.replaceAll(String regex, String replacement);

# 2. Purpose

- Replaces each substring of the original string that matches the given regular expression (regex) with the specified replacement string.
- Works with regular expressions, so it's very powerful for pattern-based replacements.

#### 3. Priority to Use in Problem Solving

- **High priority** when:
  - You need to replace multiple occurrences in a string.
  - You want to match patterns (not just exact substrings) using regex.
- Medium priority if the problem only requires replacing a fixed substring: in that case, replace() is simpler.

# 4. Core Examples (What it Takes)

- Parameters:
  - 1.  $regex \rightarrow a$  regular expression to match substrings.
  - 2. replacement → string to replace the matched substrings.
- Return Type: String (returns a new string with replacements; does not modify the original string).

#### Example 1: Replace digits with #

```
String text = "My phone number is 12345";
String result = text.replaceAll("\\d", "#"); // \d = any digit
System.out.println(result);
```

#### **Output:**

My phone number is #####

#### Example 2: Remove all spaces

```
String text = "Hello World";
String result = text.replaceAll("\\s", ""); // \s = any whitespace
System.out.println(result);
```

#### **Output:**

#### HelloWorld

#### Example 3: Replace vowels with \*

String text = "Java Programming"; String result = text.replaceAll("[aeiouAEIOU]", "\*"); // Replace vowels System.out.println(result);

#### **Output:**

J\*v\* Pr\*gr\*mm\*ng

#### 5. Summary Table

Feature Details

Method replaceAll(String regex, String

replacement)

**Purpose** Replace substrings matching regex

Parameters regex, replacement

**Return Type** String (new string)

**Priority** High for pattern-based replacements

**Example** Remove digits, spaces, vowels, custom patterns

Uses

If you want, I can also make a **mini Java program** that demonstrates **5+ different replaceAll cases** and prints the results, so you can see it in action for problem-solving scenarios.

Do you want me to do that?

# String replaceFirst(String regex, String replacement)

Sure! Let's break down the String.replaceFirst(String regex, String replacement) method in **Java** comprehensively.

# 1. Syntax

String newString = originalString.replaceFirst(String regex, String replacement);

- regex: A regular expression that specifies the pattern to be replaced.
- **replacement**: The string to replace the **first occurrence** of the regex match.

#### 2. Purpose

- Replaces **only the first substring** of the original string that matches the given regex with the specified replacement.
- Useful when you want to replace the first occurrence only, not all occurrences (for all, use replaceAll).

# 3. Priority in Problem Solving

- Medium priority:
  - Important in string manipulation problems where pattern-based replacement is required.
  - Often used in regex-based problems (e.g., formatting, text cleanup, or conditional replacements).
- Less frequently used than replace() or replaceAll() for basic replacements.

# 4. Core Examples

#### Example 1: Replace first occurrence of a word

```
public class Test {
  public static void main(String[] args) {
```

```
String str = "apple banana apple orange";
String newStr = str.replaceFirst("apple", "mango");
System.out.println(newStr);
}
```

#### **Output:**

mango banana apple orange

• Only the first "apple" is replaced.

#### **Example 2: Using regex**

```
public class Test {
   public static void main(String[] args) {
      String str = "cat bat rat";
      String newStr = str.replaceFirst(".at", "dog"); // ".at" matches first 3-letter word ending with
"at"
      System.out.println(newStr);
   }
}
```

#### **Output:**

dog bat rat

Regex .at matches the first "cat" only.

#### **Example 3: Special characters in regex**

```
public class Test {
  public static void main(String[] args) {
     String str = "price is $100 and $200";
     String newStr = str.replaceFirst("\\$\\d+", "$500");
     System.out.println(newStr);
  }
}
```

#### **Output:**

price is \$500 and \$200

• Replaces the **first occurrence** of a number with a \$ sign.

# 5. Return Type

- String → a new string with the first matching substring replaced.
- Original string remains unchanged (strings in Java are immutable).

#### 6. Print Example

String str = "hello hello world"; String result = str.replaceFirst("hello", "hi"); System.out.println(result);

#### **Output:**

hi hello world

• Demonstrates that only the first "hello" was replaced with "hi".

If you want, I can also make a **tiny table comparing replace, replaceFirst, and replaceAll** for quick problem-solving reference. This is extremely useful for coding interviews.

Do you want me to make that table?

# String strip()

Sure! Let's break down the **String.strip()** method in Java clearly and systematically:

# 1. Syntax

String strippedString = originalString.strip();

#### 2. Purpose

- Removes leading and trailing whitespace from a string.
- Unlike trim(), it also handles **Unicode whitespace characters**, not just ASCII spaces.

# 3. Priority in Problem Solving

- Medium to High:
  - Useful when reading input, parsing data, or comparing strings where unwanted spaces can cause logic errors.
- Essential in problems involving user input, text processing, or file parsing.

# 4. Core Examples (What it Takes)

- Takes no arguments.
- Operates on the string object it's called on.
- Removes whitespace from both **start** and **end**.

# 5. Return Type

String

• Returns a **new string** with leading and trailing whitespace removed.

Original string remains unchanged.

# 6. Example with Print

```
public class Main {
  public static void main(String[] args) {
    String text = " Hello World! ";

    // Using strip()
    String strippedText = text.strip();

    // Print the original and stripped string
    System.out.println("Original: "" + text + """);
    System.out.println("Stripped: "" + strippedText + """);
  }
}
```

#### **Output:**

Original: 'Hello World!'
Stripped: 'Hello World!'

# Summary:

- strip() → removes leading & trailing whitespace (including Unicode).
- Returns a new string.
- Very useful for input normalization and text comparison.

If you want, I can also create a **small table comparing strip(), trim(), and stripLeading()/stripTrailing()** so you know when to use each in problem solving. This is often asked in interviews.

Do you want me to make that?

# String stripLeading()(Java 11+)

Sure! Let's break down String.stripLeading() in Java 11+ in a structured way.

# 1. Syntax

String newString = originalString.stripLeading();

#### 2. Purpose

- Removes all leading whitespace (spaces, tabs, newlines) from the string.
- **Does not** remove trailing whitespace.
- Useful when you want to clean up user input or text data before processing.

# 3. Priority in Problem Solving

- Medium priority:
  - Use it when leading spaces can affect logic (e.g., parsing numbers, comparing strings).
  - Not as frequently needed as trim(), but safer in Java 11+ because it handles all Unicode whitespace correctly.

# 4. Core Examples

```
public class StripLeadingExample {
  public static void main(String[] args) {
    String str1 = " Hello World ";
    String str2 = "\t\n Java 11+ ";

// Using stripLeading()
    String result1 = str1.stripLeading();
```

```
String result2 = str2.stripLeading();

System.out.println("Original: "" + str1 + """);
System.out.println("After stripLeading(): "" + result1 + """);

System.out.println("Original: "" + str2 + """);
System.out.println("After stripLeading(): "" + result2 + """);
}
```

#### 5. What It Takes

- No parameters.
- Operates directly on the string object.

# 6. Return Type

• String (a **new string** with leading whitespace removed; original string is unchanged).

# 7. Sample Output

```
Original: ' Hello World '
After stripLeading(): 'Hello World '
Original: '
Java 11+ '
After stripLeading(): 'Java 11+ '
```

# Key Notes:

- stripLeading() is different from trim():
  - trim() removes both leading and trailing ASCII whitespace.

 stripLeading() only removes leading whitespace and handles all Unicode whitespace.

If you want, I can make a **tiny comparison chart for strip(), stripLeading(), stripTrailing(), and trim()** so you can know exactly when to use each in problem-solving.

Do you want me to do that?

# String stripTrailing()(Java 11+)

Absolutely! Let's break down **String.stripTrailing()** in Java 11+ in a structured way:

# 1. Syntax

public String stripTrailing()

# 2. Purpose

- stripTrailing() removes **trailing white spaces** (spaces, tabs, etc.) **from the end** of a string.
- It does not remove leading spaces.
- It is different from trim():
  - trim() removes both leading and trailing whitespaces.
  - stripTrailing() only affects the end of the string.

# 3. Priority in Problem Solving

• Medium priority in algorithmic problems.

- Useful when:
  - You want to clean up input strings.
  - You only care about trailing spaces and want to preserve leading spaces.
  - Useful in text formatting problems.

#### 4. Core Examples

- Takes no arguments.
- Returns a **new string** without trailing spaces.
- Return type: String

#### Example 1:

```
public class Main {
   public static void main(String[] args) {
      String str = " Hello World ";
      String result = str.stripTrailing();
      System.out.println("Original: "' + str + """);
      System.out.println("After stripTrailing(): "' + result + """);
   }
}
```

#### Output:

```
Original: ' Hello World ' After stripTrailing(): ' Hello World'
```

#### **Example 2 (no trailing spaces):**

```
String str = "Java 11";
System.out.println(str.stripTrailing()); // Output: 'Java 11'
```

#### Example 3 (only spaces):

```
String str = " ";
System.out.println(""" + str.stripTrailing() + """); // Output: "
```

### **✓** Summary Table:

Feature	Description
Syntax	<pre>public String stripTrailing()</pre>
Purpose	Removes trailing whitespaces from the string
Arguments	None
Return Type	String
Use Case Priority	Medium, mainly for cleaning strings
Example	" Hello ".stripTrailing() → " Hello"

If you want, I can also make a **small comparison table between trim()**, **strip()**, **stripLeading()**, **and stripTrailing()** for easier understanding in one glance.

Do you want me to do that?

# String trim()

Sure! Let's break down **String.trim()** in Java in detail as you requested:

# 1. Syntax

public String trim()

# 2. Purpose

The trim() method is used to remove leading and trailing whitespace from a string.

- Whitespace includes spaces, tabs, and other invisible characters at the start or end of the string.
- It does not remove spaces in the middle of the string.

# 3. Priority in Problem Solving

- **Medium priority**: Useful in string manipulation problems where extra spaces can cause wrong comparisons, input validation, or parsing issues.
- Commonly used in:
  - Removing unwanted spaces before storing input
  - Comparing strings accurately
  - Preparing strings for parsing or splitting

# 4. Core Example

- Takes: No parameters
- Returns: A new string with leading and trailing whitespace removed

```
public class TrimExample {
   public static void main(String[] args) {
      String str = " Hello World! "; // String with spaces at start and end
      String trimmedStr = str.trim(); // Removes leading & trailing spaces

      // Print the value
      System.out.println("Original String: "" + str + """);
      System.out.println("Trimmed String: "" + trimmedStr + """);
   }
}
```

# 5. Output

Original String: ' Hello World! ' Trimmed String: 'Hello World!'

#### Key Notes:

- trim() does not modify the original string because strings in Java are immutable.
- Always use trim() before comparing strings or splitting input data to avoid hidden whitespace errors.

If you want, I can make a **tiny table with all string methods** like trim() showing their syntax, return type, and examples for easy reference during problem solving. It'll save a lot of time.

Do you want me to do that?

# String repeat(int count) (Java 11+)

Here's a detailed breakdown of the String.repeat(int count) method in Java 11+ for problem-solving purposes:

# 1. Syntax

String repeatedString = originalString.repeat(int count);

# 2. Purpose

- The method repeats the original string count number of times.
- Useful for generating repeated patterns, padding strings, or formatting output in problem-solving.

# 3. Priority in Problem Solving

- **Medium priority:** Not used in every problem, but very handy in:
  - Pattern generation problems.
  - Constructing repeated sequences efficiently.
  - Avoiding loops for repeated string concatenation.

#### 4. Core Examples

- What it takes:
  - An integer count (>= 0) representing how many times the string should be repeated.
  - o If count is  $0 \rightarrow$  returns an empty string.
  - ∘ If count  $< 0 \rightarrow$  throws IllegalArgumentException.
- Return type: String

# 5. Example Code

```
public class RepeatExample {
    public static void main(String[] args) {
        String str = "Hi! ";

        // Repeat 3 times
        String repeated = str.repeat(3);

        // Print the result
        System.out.println(repeated);

        // Repeat 0 times
        System.out.println(str.repeat(0));

        // Example with invalid count (uncomment to see exception)
        // System.out.println(str.repeat(-1));
```

```
}
```

#### 6. Output

Hi! Hi! Hi!

(empty string)

# Key Points:

- Avoid manual loops to concatenate strings repeatedly—repeat() is more concise and readable.
- Always ensure count >= 0 to prevent exceptions.
- Useful for quick pattern generation in algorithmic problems.

If you want, I can make a **tiny table comparing repeat() with using loops and StringBuilder** for repeated string generation to see why repeat() is preferable in problem solving. Do you want me to do that?

# String[] split(String regex)

Absolutely! Let's break down the **String.split(String regex)** method in Java in a structured way:

# 1. Syntax

String[] split(String regex)

• regex: A regular expression that specifies the delimiter by which the string is split.

#### 2. Purpose

- To split a string into an array of substrings based on a specific delimiter (defined by a regex).
- Useful in problem solving when you need to break down strings for parsing, analysis, or processing.

#### 3. Priority in Problem Solving

- High Priority: Many coding problems involve parsing strings, reading CSV data, processing input, etc.
- It's essential for **string manipulation tasks** in competitive programming, interview questions, and LeetCode problems.

# 4. Core Examples

- Splitting by space, comma, or any regex.
- Can handle multiple delimiters if regex is used.

```
public class SplitExample {
   public static void main(String[] args) {
      String str = "Java,Python,C++,JavaScript";

      // Split by comma
      String[] languages = str.split(",");

      // Print each value
      for (String lang : languages) {
            System.out.println(lang);
      }

      // Another example: split by space
      String str2 = "I love coding";
      String[] words = str2.split(" ");
      for (String word : words) {
```

```
System.out.println(word);
}

// Example with regex: split by comma or space
String str3 = "Java, Python C++ JavaScript";
String[] result = str3.split("[, ]+"); // comma or space
for (String r : result) {
    System.out.println(r);
}
}
```

#### 5. What it Takes

• Input: A String containing a regular expression (regex) as a delimiter.

# 6. Return Type

• Returns a **String[]** (array of substrings).

#### 7. Print Its Value

• As shown above, you can **iterate over the array** and print each element using a loop (for loop or Arrays.toString()).

System.out.println(Arrays.toString(languages)); // Optional, prints whole array

#### **Output for first example:**

Java Python C++ JavaScript If you want, I can also make a **super concise table** for split() showing **syntax, input, output, examples, and notes** all in **one glance**—perfect for quick reference during problem solving.

Do you want me to make that table?

# String[] split(String regex, int limit)

Sure! Let's break down the **Java split(String regex, int limit) method** in detail, covering all the points you requested.

#### 1. Syntax

public String[] split(String regex, int limit)

# 2. Purpose

- Splits a string around matches of the given regular expression (regex).
- The limit parameter controls the **number of resulting substrings**:
  - o **limit > 0**  $\rightarrow$  at most limit substrings, last substring contains the rest.
  - limit = 0 → trailing empty strings are discarded.
  - $\circ$  **limit < 0**  $\rightarrow$  no limit, trailing empty strings are included.

# 3. Priority in Problem Solving

- High Priority for parsing strings, tokenization, data cleaning, and competitive programming problems that involve splitting inputs.
- Use this when you need **controlled splitting**, not just all possible splits.

# 4. Core Examples

#### Example 1: Limit > 0

```
String str = "apple,banana,orange,mango";
String[] fruits = str.split(",", 2); // Limit 2
for(String f : fruits) {
    System.out.println(f);
}
```

#### **Output:**

apple

banana,orange,mango

Explanation: Splits at most 2 times. The rest goes into the last element.

#### Example 2: Limit = 0

```
String str = "a,b,c,,";
String[] arr = str.split(",", 0);
for(String s : arr) {
    System.out.println("[" + s + "]");
}
```

#### **Output:**

- [a]
- [b]
- [c]

Explanation: Trailing empty strings are removed.

#### Example 3: Limit < 0

```
String str = "a,b,c,,";
String[] arr = str.split(",", -1);
for(String s : arr) {
    System.out.println("[" + s + "]");
}
```

#### **Output:**

[a]

[b]

[c]

Explanation: No limit, trailing empty strings are included.

# 5. Return Type

• **String[]** → an array of strings.

# 6. Printing Values

```
You can print using a loop or {\tt Arrays.toString():}
```

import java.util.Arrays;

```
String str = "apple,banana,orange,mango";
String[] fruits = str.split(",", 3);
System.out.println(Arrays.toString(fruits));
```

#### **Output:**

[apple, banana, orange,mango]

# **✓** Summary:

- Use split(regex, limit) when you need controlled splits.
- limit helps in handling trailing data or limiting array size.

 Always returns a **String array**, which you can loop through or print with Arrays.toString().

If you want, I can make a **single table** showing **regex**, **limit values**, **and outputs** for all cases—it's very handy for quick problem-solving reference.

Do you want me to make that table?

# static String join(CharSequence delimiter, CharSequence... elements)

Sure! Let's break down the String.join method in Java in a structured way as per your request.

#### 1. Syntax

static String join(CharSequence delimiter, CharSequence... elements)

- delimiter: A CharSequence (like a String) that separates each element in the final joined string.
- elements: A varargs parameter (CharSequence...) representing the elements to join.

# 2. Purpose

The join method is used to **combine multiple strings or character sequences into a single string**, with a specified delimiter between them.

It's very useful in **problem-solving** when you need to produce **formatted output**, **CSV-like strings**, or **concatenate array/list elements** efficiently.

# 3. Priority in Problem Solving

- **High priority** for:
  - Joining array or list elements into a single string.
  - Creating readable output for debugging or display.
- Not needed if you are concatenating only two strings (use + operator instead).

# 4. Core Examples (What It Takes)

#### Example 1: Joining multiple strings with a comma

```
String result = String.join(",", "Apple", "Banana", "Cherry");
System.out.println(result);
```

#### **Output:**

Apple, Banana, Cherry

### **Example 2: Joining elements from an array**

```
String[] fruits = {"Mango", "Orange", "Grapes"};
String result = String.join(" | ", fruits);
System.out.println(result);
```

#### **Output:**

Mango | Orange | Grapes

#### **Example 3: Joining list elements**

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

```
List<String> colors = List.of("Red", "Green", "Blue");
String result = String.join(" - ", colors);
System.out.println(result);
```

#### **Output:**

Red - Green - Blue

# 5. Return Type

String

• The method returns a **single string** containing all elements joined by the delimiter.

# 6. Printing Its Value

String value = String.join(";", "A", "B", "C"); System.out.println("Joined value: " + value);

#### **Output:**

Joined value: A;B;C

If you want, I can also create a **small Java snippet that demonstrates String. join with arrays, lists, and individual strings all together**, so you can see its full versatility.

Do you want me to do that?

# static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)

Sure! Let's break down the **String.join** method in Java step by step for clarity.

# 1. Syntax

public static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)

delimiter: A CharSequence (like a String) used to separate each element.

 elements: An Iterable collection of objects that implement CharSequence (like String, StringBuilder, etc.)

# 2. Purpose

- Combines the elements of an iterable (like List<String>) into a single String.
- Inserts the specified **delimiter** between each element.
- Very handy for printing lists, arrays, or collections as a single string.

# 3. Priority to Use in Problem Solving

- **High priority** when you need to convert collections or arrays into readable strings with separators.
- Often used in formatting output, joining CSV values, or preparing data for APIs.
- Saves manual iteration and string concatenation.

# 4. Core Examples

#### **Example 1: Join a List of Strings**

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry");
        String result = String.join(", ", fruits);
        System.out.println(result);
    }
}
```

#### **Output:**

#### **Example 2: Join Using StringBuilder**

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<StringBuilder> words = Arrays.asList(
            new StringBuilder("Hello"),
            new StringBuilder("World")
        );
        String result = String.join(" - ", words);
        System.out.println(result);
    }
}
```

#### **Output:**

Hello - World

#### 5. What It Takes

- CharSequence delimiter: e.g., ", ", "-", "|", " "
- Iterable<? extends CharSequence> elements: e.g., List<String>, Set<String>, LinkedList<StringBuilder>

# 6. Return Type

• Returns a String that contains all elements separated by the given delimiter.

# **7. Print Its Value**

List<String> numbers = Arrays.asList("One", "Two", "Three");

```
String joined = String.join(" | ", numbers);
System.out.println(joined);
```

#### **Output:**

One | Two | Three

If you want, I can also give a **small comparison with Collectors.joining()**, which is another popular way to join elements in Java **for problem-solving in competitive coding**. It's often faster and more flexible for streams.

Do you want me to do that?

# String toLowerCase()

Here's a complete breakdown of the String.toLowerCase() method in Java:

# 1. Syntax

String lowerCaseString = originalString.toLowerCase();

#### 2. Purpose

- Converts all characters in a string to lowercase according to the default locale.
- Useful for case-insensitive comparisons or standardizing string input.

# 3. Priority in Problem Solving

- **High** for problems where:
  - You need to compare strings ignoring case.
  - You need to normalize user input.

• Not needed if **case matters** (like passwords in some scenarios).

# 4. Core Examples

```
String str1 = "HELLO WORLD";
String str2 = "Java Programming";

// Convert to lowercase
String result1 = str1.toLowerCase();
String result2 = str2.toLowerCase();

// Print the results
System.out.println(result1); // Output: hello world
System.out.println(result2); // Output: java programming
```

- **Takes:** No parameters.
- **Returns:** A **new string** with all lowercase characters (original string remains unchanged).

# 5. Return Type

String

# Printed Values

hello world java programming

If you want, I can make a **small table of all important String methods** like this with syntax, purpose, and examples, so you can use it for **problem-solving reference**.

Do you want me to do that?

# String toLowerCase(Locale locale)

Here's a complete breakdown for the Java method **String toLowerCase(Locale locale)**:

### 1. Syntax

String lowerCaseString = originalString.toLowerCase(Locale locale);

### 2. Purpose

- Converts **all characters** of a string to **lowercase** according to the rules of the specified **Locale**.
- Useful when case sensitivity matters for locale-specific alphabets (e.g., Turkish I vs i).

### 3. Priority in Problem Solving

- **Medium priority** for algorithmic problems.
- Mostly used in:
  - Case-insensitive comparisons
  - Normalization of input strings
  - Sorting or searching strings

### 4. Core Examples & What It Takes

Parameter it takes:

• Locale locale – defines the language rules for lowercase conversion.

#### **Example 1: Basic Usage**

```
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
        String str = "HELLO WORLD";
        String lower = str.toLowerCase(Locale.ENGLISH);
        System.out.println(lower);
    }
}
```

#### **Output:**

hello world

#### **Example 2: Turkish Locale**

```
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
        String str = "I";
        String lower = str.toLowerCase(new Locale("tr", "TR"));
        System.out.println(lower);
    }
}
```

#### **Output:**

ı // Turkish lowercase dotless i

### 5. Return Type

• String – the converted lowercase string.

### 6. Printing Its Value

Already shown in the examples using:

System.out.println(lower);

### Summary:

Converts a string to lowercase according to a locale.

• Takes: Locale locale

Returns: String

• Use for case-insensitive operations, especially in multi-lingual scenarios.

If you want, I can make a **table of all Java String case methods** with their **syntax**, **return type**, **and examples** for quick reference. It'll make problem solving super fast.

Do you want me to make that?

# String to Upper Case()

Here's a detailed breakdown for the Java String.toUpperCase() method:

### 1. Syntax

String upperCaseString = originalString.toUpperCase();

### 2. Purpose

• Converts all characters in a string to uppercase.

• Useful in standardizing input, case-insensitive comparisons, or formatting output.

### 3. Priority in Problem Solving

- Medium priority:
   Important in problems involving string comparison, data normalization, or text processing.
- Not needed for numeric computations or low-level string manipulations.

### 4. Core Examples

#### What it takes:

- It is called on a **String object**.
- Does not take any arguments.

#### Example:

String name = "Routh Kiran Babu"; String upperName = name.toUpperCase();

### 5. Return Type

- String
- Returns a **new string** with all characters converted to uppercase.
- Original string remains unchanged (strings are immutable in Java).

#### 6. Print Its Value

System.out.println(upperName); // Output: ROUTH KIRAN BABU

### ✓ Summary Table

Feature	Details
Syntax	str.toUpperCase()
Purpose	Convert string to uppercase
Priority	Medium (useful for string comparison/formatting)
Parameters	None
Return Type	String
Example	"hello".toUpperCase() $\rightarrow$ "HELLO"

If you want, I can make a small Java code snippet demonstrating toUpperCase() with multiple examples and printing outputs for better clarity.

Do you want me to do that?

# String to Upper Case (Locale locale)

Here's a complete breakdown for the Java method **String toUpperCase(Locale locale)**:

### 1. Syntax

String upperCaseString = originalString.toUpperCase(Locale locale);

### 2. Purpose

• Converts all characters of a string to **uppercase** using the rules of a specific **Locale**.

- Useful when handling locale-specific text (like Turkish "i" vs "İ").
- Unlike toUpperCase() without a locale, this ensures proper behavior for internationalization.

### 3. Priority in Problem Solving

- Medium priority in most algorithmic problems.
- Use it when:
  - You need to normalize string comparisons ignoring case.
  - Working with internationalized text.
- Less priority for pure numeric or logical problems.

### 4. Core Example – What It Takes

- Parameter: Locale locale a Locale object specifying the language/country rules.
- **Return Type:** String the uppercase version of the original string.

### 5. Example Code

```
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
        String original = "hello world";

        // Convert to uppercase using default locale (US)
        String upperUS = original.toUpperCase(Locale.US);
        System.out.println("Uppercase (US Locale): " + upperUS);

        // Convert to uppercase using Turkish locale
```

```
String upperTR = original.toUpperCase(new Locale("tr", "TR"));
System.out.println("Uppercase (Turkish Locale): " + upperTR);
}
```

### 6. Output

Uppercase (US Locale): HELLO WORLD Uppercase (Turkish Locale): HELLO WORLD

Note: For Turkish, letters like i behave differently in uppercase. So using the correct Locale ensures accuracy.

If you want, I can make a **small table showing all toUpperCase variants**, **parameters**, **return types**, **and examples** for super quick reference. This is handy for problem solving.

Do you want me to do that?

# byte[] getBytes()

Sure! Let's break down the byte[] getBytes() method in Java step by step, covering all you asked.

### 1. Syntax

byte[] byteArray = stringObject.getBytes();

Here, stringObject is any instance of the String class.

### 2. Purpose

• The getBytes() method converts a String into a sequence of bytes.

- It's commonly used when you need to store strings in binary form, write to files, or send over a network.
- By default, it uses the platform's default charset (e.g., UTF-8).

### 3. Priority in Problem Solving

- Moderate priority: Not every problem needs it.
- Use it when:
  - You want byte-level manipulation of strings.
  - You are working with I/O operations (files, sockets).
- Rarely used in typical algorithmic problems unless **encoding or byte-level comparison** is needed.

### 4. Core Examples

### **Example 1: Basic Usage**

```
String str = "Hello";
byte[] bytes = str.getBytes();
for (byte b : bytes) {
    System.out.print(b + " ");
}
```

#### Output (platform dependent, UTF-8 usually):

72 101 108 108 111

#### **Explanation:**

• Each character is converted to its ASCII/UTF-8 numeric value.

```
\circ H = 72, e = 101, 1 = 108, o = 111.
```

#### **Example 2: Specifying Charset**

import java.nio.charset.StandardCharsets;

```
String str = "Hello";
byte[] bytes = str.getBytes(StandardCharsets.UTF_8);
```

System.out.println(java.util.Arrays.toString(bytes));

#### **Output:**

```
[72, 101, 108, 108, 111]
```

#### Why use charset?

- Makes your program platform-independent.
- Recommended when exchanging data across different systems.

#### 5. What It Takes

- It does not take any arguments in its basic form.
- Optionally, it can take a **Charset** or **String encoding**:

byte[] bytes = str.getBytes("UTF-8");

### 6. Return Type

byte[]

• Returns a byte array representing the string content.

### 7. Printing Its Value

```
String str = "Hello World";
byte[] bytes = str.getBytes();

// Print using Arrays.toString()
System.out.println(java.util.Arrays.toString(bytes));
```

#### **Output:**

```
[72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]
```

### **Explanation:**

- Each character in "Hello World" is converted to its **byte value**.
- 32 is the ASCII code for **space**.

### Summary Table

Feature	Details
Syntax	<pre>byte[] bytes = str.getBytes();</pre>
Purpose	Convert String $\rightarrow$ byte array
Priority	Moderate (use for I/O, encoding, byte-level ops)
Takes	Optional Charset (getBytes("UTF-8"))
Returns	byte[]
Example Output	[72, 101, 108, 108, 111]

If you want, I can also **show a small trick to convert the byte[] back to a string** and print it in readable form, which is often useful in problem-solving.

# byte[] getBytes(Charset charset)

Sure! Let's break down everything about the Java method byte[] getBytes(Charset charset) in a clear and structured way:

### 1. Syntax

byte[] byteArray = string.getBytes(Charset charset);

#### **Explanation:**

- string → The string you want to convert.
- Charset charset → The character set used for encoding (like StandardCharsets.UTF\_8, StandardCharsets.ISO\_8859\_1).
- Returns a **byte array** (byte[]) representing the encoded string.

### 2. Purpose

- Converts a String into a byte array using a specific character encoding.
- Useful in **networking**, **file operations**, **encryption**, **hashing**, or any situation requiring byte-level manipulation.

### 3. Priority in Problem Solving

- **Medium to high priority** depending on the context:
  - **High:** When solving problems related to **encoding, decoding, or binary data** (e.g., cryptography, file I/O).

Medium: In general string manipulations, sometimes default getBytes()
 (without charset) is used, but specifying a charset avoids platform-dependent issues.

### 4. Core Examples

```
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String str = "Hello World";

        // Convert String to byte array using UTF-8
        byte[] utf8Bytes = str.getBytes(StandardCharsets.UTF_8);
        System.out.println("UTF-8 bytes: " + Arrays.toString(utf8Bytes));

        // Convert String to byte array using ISO-8859-1
        byte[] isoBytes = str.getBytes(StandardCharsets.ISO_8859_1);
        System.out.println("ISO-8859-1 bytes: " + Arrays.toString(isoBytes));
    }
}
```

#### 5. Takes

• Parameter: Charset charset

```
Example values:
```

- StandardCharsets.UTF\_8
- StandardCharsets.ISO\_8859\_1
- o Charset.forName("UTF-16")

### 6. Return Type

byte[] → An array of bytes representing the string in the specified encoding.

### 7. Printing Its Value

• Use Arrays.toString(byteArray) to print a byte array in a readable format. Example:

byte[] bytes = "Hello".getBytes(StandardCharsets.UTF\_8);
System.out.println(Arrays.toString(bytes));
// Output: [72, 101, 108, 108, 111]

Each number represents the ASCII/UTF-8 value of the character.

### ✓ Summary Table

Feature	Description
Syntax	string.getBytes(Charset charset)
Purpose	Convert a string to a byte array using a specific charset
Parameter	Charset charset (e.g., StandardCharsets.UTF_8)
Return Type	byte[]
Priority	Medium-High (important in encoding/decoding, file/network operations)
Print Example	Arrays.toString(byteArray)

If you want, I can also create a **small table showing some strings and their UTF-8 vs ISO-8859-1 byte arrays**, which helps in **problem-solving comparisons**.

Do you want me to do that?

# byte[] getBytes(String charsetName)

Sure! Let's break down the Java method byte[] getBytes(String charsetName) step by step. I'll cover syntax, purpose, priority, examples, return type, and how to print it.

### 1 Syntax

byte[] byteArray = stringObject.getBytes(String charsetName) throws
UnsupportedEncodingException;

- stringObject → The String instance whose bytes you want to get.
- charsetName → Name of the charset (like "UTF-8", "ISO-8859-1", "US-ASCII").

### 2 Purpose

- Converts a **Java String into a byte array** using the specified character encoding.
- Useful when you need to send strings over network, write to files, or perform encoding/decoding.

### **3** Priority in Problem Solving

- **Medium priority** in general algorithmic problems (like LeetCode) because most problems don't require manual encoding.
- **High priority** in problems involving:
  - File I/O
  - Network communication
  - Encoding/decoding

### 4 Core Examples (Input it Takes & Return Type)

- Input: A String and a valid charset name (UTF-8, ISO-8859-1)
- Return Type: byte[]

#### **Example 1: Basic Usage**

```
import java.io.UnsupportedEncodingException;
import java.util.Arrays;

public class GetBytesExample {
    public static void main(String[] args) {
        try {
            String str = "Hello World!";
            byte[] bytes = str.getBytes("UTF-8"); // convert to UTF-8 byte array

            // Print the byte array
            System.out.println("Byte array: " + Arrays.toString(bytes));
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
    }
}
```

#### **Output:**

Byte array: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]

#### **Example 2: Different Charset**

```
String str = "Hello World!";
byte[] asciiBytes = str.getBytes("US-ASCII");
System.out.println(Arrays.toString(asciiBytes));
```

• Returns the same byte values as ASCII codes of characters.

### **5** Printing Its Value

- Since byte[] is an array, directly printing it gives a memory reference.
- Use Arrays.toString(byteArray) to display readable bytes.

System.out.println(Arrays.toString(byteArray));

### Summary Table

Feature	Details
Syntax	<pre>byte[] bytes = string.getBytes(String charsetName)</pre>
Purpose	Convert String to byte[] with specific encoding
Return Type	byte[]
Input	String + charset name (UTF-8, ISO-8859-1)
Use in problem solving	Medium to High (for encoding/network/crypto problems)
Printing	Arrays.toString(byteArray)

If you want, I can also show a **quick trick to convert it back to String from bytes**, which is often used in problem-solving for encoding-decoding questions.

Do you want me to show that?

# void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

Sure! Let's break down the getChars method from Java's String class in detail:

#### 1. Syntax

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

#### 2. Purpose

- The getChars() method copies a substring from a String into a character array (char[]).
- It does not return a value because it directly modifies the target array.
- Useful when you need characters from a string for further processing or manipulation.

### 3. Priority in Problem Solving

- **Medium priority** in problem-solving.
- Use it when:
  - You need high-performance character extraction.
  - You want to avoid creating new String objects.
  - You are doing array-based algorithms with strings (like substring manipulations, custom parsing, or in-place modifications).

### 4. Parameters (What it Takes)

Parameter	Description
srcBegin	Start index (inclusive) in the string.
srcEnd	End index (exclusive) in the string.
dst	Destination char [] array where characters will be copied.

dstBegin Start index in the destination array to begin copying.

- Important: Indexing is 0-based.
- Throws IndexOutOfBoundsException if indexes are invalid.

### 5. Return Type

- void → It does not return any value.
- The copied characters are placed into the provided char [].

### 6. Core Example

```
public class GetCharsExample {
   public static void main(String[] args) {
      String str = "HelloWorld";
      char[] charArray = new char[5];

   // Copy "World" into charArray starting at index 0
      str.getChars(5, 10, charArray, 0);

   // Print the character array
      System.out.println(charArray); // Output: World
   }
}
```

#### **Explanation:**

- srcBegin = 5, srcEnd = 10 → takes substring "World" from "HelloWorld".
- dst = charArray → copy characters into this array.
- dstBegin =  $0 \rightarrow$  start placing characters at index 0 of charArray.

System.out.println(charArray) prints "World".

### ✓ Key Points to Remember

- 1. getChars does not return a string, it fills an array.
- 2. Indices must be carefully handled; otherwise, you get IndexOutOfBoundsException.
- 3. Fast and memory-efficient compared to substring().toCharArray().

If you want, I can also make a **visual diagram** showing exactly how the characters are copied from String to the char[], which makes understanding srcBegin, srcEnd, and dstBegin super intuitive.

Do you want me to do that?

# char[] toCharArray()

Sure! Let's break down everything about the Java toCharArray() method of the String class in a structured way.

### 1 Syntax

char[] charArray = stringVariable.toCharArray();

### 2 Purpose

- Converts a **String** into a **char array** (char[]), where each element of the array is a character from the string.
- Useful when you want to:
  - Traverse a string character by character.

o Perform character-level operations like counting, replacing, or reversing.

### 3 Priority in Problem Solving

- **High priority** when solving problems that involve:
  - o Character manipulation.
  - o Palindrome checking.
  - Frequency counting of characters.
  - Reversing a string manually.
- Medium/Low priority if you can work with charAt() in loops directly.

### 4 Core Examples

- What it takes: No arguments (empty parentheses).
- Return type: char[]

#### **Example 1: Basic Usage**

```
public class Main {
   public static void main(String[] args) {
      String str = "Hello";
      char[] charArray = str.toCharArray(); // Convert String to char array
      // Print the char array
      for (char c : charArray) {
            System.out.print(c + " ");
      }
    }
}
```

#### **Output:**

### **Example 2: Using in Problem Solving**

```
public class Main {
  public static void main(String[] args) {
    String str = "Level";
    char[] charArray = str.toCharArray();

    // Check if palindrome
    boolean isPalindrome = true;
    for (int i = 0; i < charArray.length / 2; i++) {
        if (charArray[i] != charArray[charArray.length - 1 - i]) {
            isPalindrome = false;
            break;
        }
    }
    System.out.println("Is palindrome? " + isPalindrome);
}</pre>
```

#### **Output:**

Is palindrome? true

### ✓ Summary Table

Feature	Details
Syntax	<pre>char[] charArray = str.toCharArray();</pre>
Purpose	Convert String to char array for character-level operations
Takes	No arguments
Returns	char[]
Priority	High for string manipulation, medium if direct charAt() works

If you want, I can also give a **shortcut one-liner** to print the char array directly without loops. It's very handy for coding problems.

Do you want me to show that?

# static String valueOf(char[] data)

Sure! Let's break down the Java method static String valueOf(char[] data) in a structured way for clarity:

### 1. Syntax

String str = String.valueOf(char[] data);

### 2. Purpose

- Converts a char array into a String.
- It is a **static method** of the String class.
- Useful when you have characters stored in an array and want them as a single String.

### 3. Priority in Problem Solving

- **High priority** for problems involving string construction or manipulation.
- Often used in:
  - Converting char arrays from input to Strings.
  - Efficient string creation from characters.

Situations where concatenation of characters is needed.

### 4. Core Example

```
What it takes: a char array
Return type: String

public class Main {
    public static void main(String[] args) {
        char[] data = {'H', 'e', 'l', 'l', 'o'};

    // Using valueOf to convert char array to String
    String str = String.valueOf(data);

    // Printing the value
    System.out.println("String value: " + str);
    }
}
```

### 5. Output

String value: Hello

### ✓ Summary:

• Takes: char[] data

• Returns: String

• Use Case: Convert character arrays to Strings efficiently.

If you want, I can also make a **table of all similar String.valueOf() overloaded methods** in Java with examples for quick reference during problem solving.

Do you want me to make that table?

# static String valueOf(char[] data, int offset, int count)

Sure! Let's break down the String.valueOf(char[] data, int offset, int count) method in **Java** in detail:

### 1. Syntax

static String valueOf(char[] data, int offset, int count)

### 2. Purpose

- Converts a subarray of characters (char[]) into a String.
- offset specifies the **starting index** in the array.
- count specifies the **number of characters** to use starting from the offset.
- Useful when you want only a part of a character array as a String instead of the whole array.

### 3. Priority in Problem Solving

- Medium to High priority for problems involving:
  - Substrings extraction from character arrays.
  - Custom string manipulations.
  - Converting portions of char arrays to strings efficiently.

#### 4. Parameters

Parameter Type

**Description** 

```
data char[] The character array to convert.

offset int The starting index in the array.

count int The number of characters to convert.
```

### 5. Return Type

#### String

 Returns a new String containing characters from data[offset] to data[offset + count - 1].

### 6. Core Examples

```
public class Main {
  public static void main(String[] args) {
     char[] data = {'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'};

  // Convert entire array
  String full = String.valueOf(data, 0, data.length);
  System.out.println("Full array: " + full);

  // Convert partial array (offset = 0, count = 5)
  String part1 = String.valueOf(data, 0, 5);
  System.out.println("Partial 1: " + part1);

  // Convert partial array (offset = 5, count = 5)
  String part2 = String.valueOf(data, 5, 5);
  System.out.println("Partial 2: " + part2);
  }
}
```

### 7. Output

Full array: HelloWorld

Partial 1: Hello Partial 2: World

### Key Notes:

- If offset or count is invalid (negative or beyond array length), it throws
   IndexOutOfBoundsException.
- More efficient than manually looping and concatenating characters to a String.

If you want, I can also make a **quick diagram showing how offset and count extract characters** from an array—it makes understanding this method super easy.

Do you want me to do that?

# static String valueOf(boolean b)

Absolutely! Let's break down the static String valueOf(boolean b) method from Java's String class in a detailed and structured way.

### 1 Syntax

String result = String.valueOf(boolean b);

### 2 Purpose

- Converts a boolean value (true or false) into its corresponding String representation.
- Useful when you need to work with boolean values as strings (e.g., for printing, concatenation, or passing to methods that require Strings).

### Priority in Problem Solving

- Medium priority.
- Essential when:
  - You need to convert booleans to strings explicitly.
  - You're **building string outputs dynamically** from boolean conditions.
- Not critical for logic itself but useful for formatting output.

### Core Example (What it Takes)

- Input parameter:
  - A boolean value (true or false)
- Return type:
  - String ("true" or "false")

#### Example:

```
public class Main {
  public static void main(String[] args) {
    boolean flag1 = true;
    boolean flag2 = false;

  // Using String.valueOf(boolean)
    String str1 = String.valueOf(flag1);
    String str2 = String.valueOf(flag2);

  // Print values
    System.out.println("String representation of flag1: " + str1);
    System.out.println("String representation of flag2: " + str2);
  }
}
```

### **5** Output

String representation of flag1: true String representation of flag2: false

### **W** Key Notes:

- String.valueOf(boolean) is **safer than Boolean.toString()** in the sense that it never throws NullPointerException (though boolean primitives cannot be null anyway).
- Often used in **string concatenation**, e.g.:

System.out.println("The condition is " + String.valueOf(flag1));

If you want, I can create a **small table for all String.valueOf() overloaded methods**, showing exactly what each takes and returns — this is super useful for competitive programming and interview prep.

Do you want me to do that?

# static String valueOf(char c)

Sure! Let's break down everything about the static String valueOf(char c) method in Java in a structured way.

### 1. Syntax

String str = String.valueOf(char c);

#### 2. Purpose

- Converts a single char into a String.
- Useful when you need to manipulate characters as strings, concatenate with other strings, or return strings from methods that originally produce a char.

### 3. Priority to Use in Problem Solving

- **High**: Whenever you need to convert a char to String for string operations (like concatenation, substring operations, or printing).
- Preferred over concatenating " " + c because it is explicit and more readable.

### 4. Core Examples

#### **Example 1: Basic usage**

```
char ch = 'A';
String str = String.valueOf(ch);
System.out.println(str); // Output: A
```

#### **Example 2: Using in concatenation**

```
char ch = 'Z';
String message = "Character is: " + String.valueOf(ch);
System.out.println(message); // Output: Character is: Z
```

#### 5. What It Takes

- Parameter: A single char (char c)
- Example: 'a', '1', '@'

### 6. Return Type

• String: Returns a string representation of the given character.

#### 7. Print Its Value

char ch = 'X';

System.out.println(String.valueOf(ch)); // Output: X

### **Summary**:

- String.valueOf(char c) is the standard way to convert a char into a String.
- Always returns a String.
- Very commonly used in string manipulation problems.

If you want, I can make a **tiny Java snippet that shows valueOf with multiple chars and prints them together**, which is often how it's used in problem-solving.

Do you want me to do that?

# static String valueOf(int i)

Sure! Let's break down the static String valueOf(int i) method from Java's String class in a clear and structured way:

### 1. Syntax

String str = String.valueOf(int i);

### 2. Purpose

The purpose of String.valueOf(int i) is to convert an integer (int) into its corresponding String representation.

Example: converting 123 (int) → "123" (String)

• This is particularly useful when you need to concatenate integers with strings or pass integers to methods that require strings.

### 3. Priority in Problem Solving

- High priority in problems involving string manipulation, concatenation, or formatting outputs.
- Often used in **competitive programming**, **LeetCode**, **or algorithm problems** where numeric results need to be converted to strings for further processing.
- Alternative: Integer.toString(int i) does a similar conversion.

### 4. Core Example

```
public class Main {
    public static void main(String[] args) {
        int number = 456;

        // Convert int to String
        String strNumber = String.valueOf(number);

        // Print the value
        System.out.println("String value: " + strNumber);

        // Check type (optional)
        System.out.println("Type: " + ((Object)strNumber).getClass().getSimpleName());
    }
}
```

#### **Output:**

String value: 456 Type: String

#### 5. What It Takes

- Parameter: an int value
- **Example:** valueOf(100) or valueOf(-50)

### 6. Return Type

- Return Type: String
- The returned string is the string representation of the given integer.

### Key Notes:

- valueOf() is overloaded: it can also take double, float, long, boolean, char,
   Object, etc., and return their string equivalents.
- Always prefer String.valueOf(int) over manual concatenation (like " " + i) for clarity and readability.

If you want, I can also create a **tiny table of all String.valueOf() overloads with examples** so it's super easy for problem-solving. Do you want me to do that?

# static String valueOf(long I)

Sure! Let's break down **static String valueOf(long 1)** from Java step by step in a clear and structured way.

### 1 Syntax

String result = String.valueOf(long I);

### 2 Purpose

The valueOf(long 1) method is used to convert a long type value into its String representation.

It is a convenient way to transform numeric data into strings for display, concatenation, or further string operations.

### Priority to Use in Problem Solving

- ✓ High priority often used in problems where:
  - You need to concatenate numbers with strings.
  - You need numeric values as strings for output formatting.
  - You are converting data types in string-based algorithms.

It's **safer and more concise** than manually concatenating numbers with an empty string (1 + "") because it clearly indicates the intention of conversion.

### **4** Core Example

```
public class ValueOfExample {
   public static void main(String[] args) {
      long num = 123456789L;

      // Using valueOf to convert long to String
      String strNum = String.valueOf(num);

      // Printing the type and value
      System.out.println("Value: " + strNum);
      System.out.println("Type: " + ((Object)strNum).getClass().getSimpleName());
    }
}
```

### **5** What It Takes (Parameter)

• Parameter: a single long value

Example: 12345L

### 6 Return Type

Return Type: String

• It returns a string representation of the long value passed.

### **7**Output of the Example

Value: 123456789 Type: String

If you want, I can make a **tiny table comparing valueOf for all primitive types** in Java with examples. This is **super useful for problem-solving and interviews**.

Do you want me to make that?

# static String valueOf(float f)

Sure! Let's break down **static String valueOf(float f)** from Java's String class in detail:

### 1 Syntax

String str = String.valueOf(float f);

### 2 Purpose

• Converts a **float** value into its **String representation**.

• Useful when you need to combine numeric values with text or store numeric values as strings.

### 3 Priority in Problem Solving

- **High priority** for problems involving:
  - Concatenation of numbers and strings.
  - Displaying float values in UI or logs.
  - Passing numeric data as strings to functions or APIs.
- Less relevant for numeric-only calculations; mainly for **conversion and representation**.

### 4 Core Examples

#### **Example 1: Simple float to String**

float num = 12.34f;

String str = String.valueOf(num);

System.out.println(str); // Output: "12.34"

#### **Example 2: Using in concatenation**

float price = 99.99f;

String message = "The price is: " + String.valueOf(price);

System.out.println(message); // Output: "The price is: 99.99"

#### **Example 3: Storing float as String**

float pi = 3.14159f;

String piStr = String.valueOf(pi);

System.out.println(piStr); // Output: "3.14159"

### **5** What it takes

#### • Parameter:

float  $f \rightarrow$  the float value you want to convert.

### 6 Return Type

• **String** → representation of the float value.

### 7 Printing its value

float f = 45.67f;

System.out.println(String.valueOf(f)); // Output: "45.67"

### **Summary**:

Input: float

• Output: String

• **Use Case:** Conversion from float → String for display or concatenation

• **Priority:** High for string manipulation involving floats

If you want, I can also create a **small Java program showing valueOf usage with float, int, double, boolean all together**, so you can compare them. Do you want me to do that?

# static String valueOf(double d)

Sure! Let's break down the **static String valueOf(double d)** method in Java step by step.

### 1. Syntax

String result = String.valueOf(double d);

## 2. Purpose

The String.valueOf(double d) method is used to convert a double value into its corresponding String representation.

This is very useful when you want to concatenate numbers with strings or print numbers as strings.

## 3. Priority in Problem Solving

- **High Priority** in problems where **number-to-string conversion** is needed.
- Useful in:
  - Concatenating numbers with text.
  - o Preparing numbers for string-based algorithms.
  - Formatting output for display or storage.

## 4. Core Examples (What It Takes)

- Takes a **primitive double value** as an argument.
- Can handle normal, scientific, and special values like NaN and Infinity.

```
double num1 = 123.45;
double num2 = -0.987;
double num3 = Double.NaN;
double num4 = Double.POSITIVE_INFINITY;
```

## 5. Return Type

- Returns: String
- The returned string is the **text representation of the double value**.

## 6. Example Code and Printing Value

```
public class Main {
   public static void main(String[] args) {
      double num = 123.456;

      // Convert double to String
      String strNum = String.valueOf(num);

      // Print the string value
      System.out.println("String representation: " + strNum);
    }
}
```

#### **Output:**

String representation: 123.456

If you want, I can also make a **table of String.valueOf()** for all primitive types with examples and outputs—it's super handy for problem-solving in Java.

Do you want me to make that?

# static String valueOf(Object obj)

Sure! Let's break down the **static String valueOf(Object obj)** method in Java in detail:

## 1. Syntax

String str = String.valueOf(obj);

- obj → can be any object (including null)
- str → resulting string

## 2. Purpose

- Converts any object into its String representation.
- If obj is null, it returns the string "null" instead of throwing an exception.
- Useful when you want to **safely convert objects to strings** for printing, logging, or concatenation.

#### 3. Priority to Use in Problem Solving

- **High priority** when you need to:
  - 1. Convert objects (Integer, Double, Character, Boolean, etc.) to strings.
  - 2. Avoid NullPointerException when converting null.
  - 3. Use objects in string concatenation or formatted output.
- Preferred over obj.toString() if obj might be null.

## 4. Core Examples

#### **Example 1: Integer**

Integer num = 100; String strNum = String.valueOf(num); System.out.println(strNum); // Output: "100"

#### **Example 2: Boolean**

Boolean flag = true;

```
String strFlag = String.valueOf(flag);
System.out.println(strFlag); // Output: "true"
```

#### **Example 3: Null Object**

```
Object obj = null;
String strObj = String.valueOf(obj);
System.out.println(strObj); // Output: "null"
```

## **Example 4: Custom Object**

```
class Person {
    String name;
    Person(String name) { this.name = name; }
    public String toString() { return "Person: " + name; }
}

Person p = new Person("Routh");
String strPerson = String.valueOf(p);
System.out.println(strPerson); // Output: "Person: Routh"
```

## 5. Return Type

public static String valueOf(Object obj)

Returns a String representation of the given object.

## 6. Printing its Value

```
Object obj = 45.67;
System.out.println(String.valueOf(obj)); // Output: "45.67"
```

## Summary

- Converts any object to a string safely.
- Handles null gracefully.

Essential for problem-solving when combining objects and strings or logging.

If you want, I can make a single neat Java program showing all the above examples together with their outputs for easy reference. Do you want me to do that?

# static String copyValueOf(char[] data)

Sure! Let's break down the copyValueOf method in Java step by step, exactly as you requested.

## 1. Syntax

static String copyValueOf(char[] data)

Optional overloaded version:

static String copyValueOf(char[] data, int offset, int count)

## 2. Purpose

The copyValueOf method is used to **convert a character array (char[]) into a String**. It essentially creates a string representation of the given character array.

- data → the character array to convert.
- offset → the starting position in the array.
- count → the number of characters to include in the string (used in the overloaded version).

## 3. Priority in Problem Solving

• Moderate Priority in problem solving:

- Useful when you have data in char[] and need to process it as a String.
- Often used in string manipulations, formatting, or preparing character arrays for comparison.
- Less commonly used than new String(char[]), but can be handy for readability and direct conversion.

## 4. Core Examples

```
public class CopyValueOfExample {
   public static void main(String[] args) {
      char[] chars = {'J', 'a', 'v', 'a'};

      // Using copyValueOf
      String str = String.copyValueOf(chars);
      System.out.println("String from char array: " + str);

      // Using overloaded version
      String str2 = String.copyValueOf(chars, 1, 2); // takes 'a' and 'v'
      System.out.println("Substring from char array: " + str2);
    }
}
```

#### 5. What it takes

Parameters:

```
    char[] data → mandatory.
    int offset, int count → optional for selecting part of array.
```

• Data Type of input: char[]

## 6. Return Type

String → a new string that represents the characters in the array.

## 7. Printing Its Value

Output from the example above:

String from char array: Java Substring from char array: av

## **Summary:**

copyValueOf is a clean way to convert character arrays to strings. It's especially useful if you want to convert the entire array or a portion of it into a readable string.

If you want, I can also make a **tiny table comparing copyValueOf vs new** String(char[]) vs valueOf(char[]), which is great for interview clarity.

Do you want me to do that?

# static String copyValueOf(char[] data, int offset, int count)

Sure! Let's break down the String.copyValueOf(char[] data, int offset, int count) method in Java in a structured way:

## 1. Syntax

public static String copyValueOf(char[] data, int offset, int count)

## 2. Purpose

 The method returns a String that represents a subarray of characters from the given character array.

- offset → starting index in the array.
- count → number of characters to include in the string.
- It is useful when you want to convert part of a char array into a string.

## 3. Priority in Problem Solving

- **Medium priority**: Not very common in competitive programming but useful when:
  - You need to extract a substring from a character array.
  - You're working with low-level character manipulations instead of full strings.
- Often, developers prefer new String(char[], offset, count) or String.valueOf(char[], offset, count) (both are equivalent).

## 4. Core Examples

```
public class Main {
   public static void main(String[] args) {
      char[] letters = {'J', 'a', 'v', 'a', 'R', 'o', 'u', 't', 'h'};

      // Extract part of the char array
      String part = String.copyValueOf(letters, 0, 4); // from index 0, take 4 chars
      System.out.println("Subarray as String: " + part); // Output: Java

      // Another example
      String anotherPart = String.copyValueOf(letters, 4, 5); // from index 4, take 5 chars
      System.out.println("Subarray as String: " + anotherPart); // Output: Routh
   }
}
```

#### 5. What it Takes

• Parameters:

- 1. char[] data  $\rightarrow$  the array of characters.
- 2. int offset  $\rightarrow$  starting index in the array.
- 3. int count  $\rightarrow$  number of characters to include.

## 6. Return Type

• Returns a String representing the specified portion of the char array.

#### 7. Printing Its Value

From the above example:

System.out.println(String.copyValueOf(letters, 0, 4)); // prints "Java" System.out.println(String.copyValueOf(letters, 4, 5)); // prints "Routh"

## Summary:

- Converts part of a char array to String.
- Useful for array-to-string manipulations.
- Return type is String.
- Equivalent to String.valueOf(char[], offset, count).

If you want, I can make a **tiny diagram showing how offset and count work visually** on the char array—it makes this concept super clear for problem solving. Do you want me to do that?

## String intern()

Sure! Let's break down everything about the **String.intern()** method in Java in a structured way:

## 1. Syntax

public String intern()

• **Method type:** Instance method

• Package: java.lang.String

• Parameters: None

#### 2. Purpose

- The intern() method returns a canonical representation of the string object.
- If the string is already in the **string pool**, it returns the reference from the pool.
- If the string is not in the pool, it **adds it to the pool** and returns that reference.

**Key point:** It helps **save memory** by using shared references from the string pool.

## 3. Priority to use in problem solving

- Low priority in competitive programming or basic problem solving unless:
  - You need reference comparison (==) instead of .equals().
  - You are trying to optimize memory by storing many duplicate strings.
- Most of the time, .equals() is enough for string comparisons in problems.

#### 4. Core Examples & What it Takes

The intern() method takes no arguments.

#### **Example 1: Basic Usage**

```
public class Main {
  public static void main(String[] args) {
    String s1 = new String("hello"); // creates object in heap
    String s2 = "hello"; // literal, stored in string pool

    System.out.println(s1 == s2); // false (different references)
    s1 = s1.intern(); // now s1 points to string pool
    System.out.println(s1 == s2); // true (same reference)
  }
}
```

#### **Output:**

false true

#### **Explanation:**

- Before intern(): s1 points to heap object, s2 points to string pool.
- After intern(): s1 now points to the **string pool object**, so s1 == s2 is true.

## 5. Return Type

• String — returns a reference to a string from the string pool.

### 6. Printing its Value

String str = new String("world");

String internedStr = str.intern(); System.out.println(internedStr); // prints "world"

#### Output:

world

## Summary Table:

Feature Details

Syntax public String intern()

Purpose Returns canonical reference from string pool

Parameters None

Return Type String

Priority Use Low, only for memory optimization or == comparisons

Core Convert heap string to pool reference for ==

Example comparison

Printing Value System.out.println(s.intern())

If you want, I can also give a more advanced example showing memory optimization with multiple strings, which is how intern() can really be useful in large-scale problem solving.

Do you want me to show that?

# static String format(String format, Object... args)

Sure! Let's break down the **String.format**(**String format**, **Object... args**) method in Java in detail:

## 1. Syntax

String formattedString = String.format(String format, Object... args);

- format: A **format string** containing placeholders (like %s, %d, %f).
- args: The values to replace the placeholders. It's a **varargs parameter**, so you can pass multiple values.

## 2. Purpose

- To create a formatted string using a template.
- Helps control how data appears (like decimal places, padding, alignment).
- Avoids manual string concatenation in complex outputs.

## 3. Priority in Problem Solving

- High for display/output formatting.
- Useful in competitive programming, report generation, or clean console output.
- Less important for algorithmic logic itself, but very important when **presenting results** neatly.

## 4. Core Examples (What it Takes)

#### Placeholders:

Placeholde r	Description
%s	String
%d	Integer
%f	Floating-point (decimal)
%c	Character

#### **Examples:**

```
// Example 1: String
String name = "Routh";
String greeting = String.format("Hello, %s!", name);
System.out.println(greeting); // Output: Hello, Routh!

// Example 2: Integer
int age = 25;
String info = String.format("I am %d years old.", age);
System.out.println(info); // Output: I am 25 years old.

// Example 3: Floating point
double price = 123.456;
String formattedPrice = String.format("Price: %.2f", price); // 2 decimal places
System.out.println(formattedPrice); // Output: Price: 123.46

// Example 4: Multiple values
String details = String.format("Name: %s, Age: %d, Score: %.1f", "Routh", 25, 98.5);
System.out.println(details); // Output: Name: Routh, Age: 25, Score: 98.5
```

#### 5. Return Type

- String
- The method returns a formatted string but does not print it. You need System.out.println() to display.

## 6. Printing Its Value

String formatted = String.format("Hello %s, your balance is \$%.2f", "Routh", 2500.567); System.out.println(formatted); // Output: Hello Routh, your balance is \$2500.57



- Use String.format() for clean, readable, and precise output formatting.
- Always remember it **returns a String**, so you need to print it if you want to see it.
- Very flexible with various placeholders and varargs values.

If you want, I can make a **small table of all the most commonly used placeholders with examples** for **quick reference in problem solving**. It's very handy during contests or coding interviews. Do you want me to do that?

# static String format(Locale I, String format, Object... args)

Sure! Let's break down the static String format(Locale 1, String format, Object... args) method in Java step by step:

## 1. Syntax

String formattedString = String.format(Locale locale, String format, Object... args);

## 2. Purpose

- It returns a formatted string using the specified locale.
- Useful when you want to format numbers, dates, or strings according to locale-specific rules.
- Acts like printf, but instead of printing, it returns the formatted string.

#### 3. Priority in Problem Solving

• High priority if you need clean string formatting.

- Especially useful in:
  - Displaying numbers with precision.
  - o Formatting currency, dates, or percentages.
  - Aligning text or padding output.
- Less priority for simple concatenation; high priority for readable, locale-aware output.

## 4. Core Examples

```
import java.util.Locale;
public class Main {
  public static void main(String[] args) {
     // Example 1: Formatting numbers with locale
     double value = 12345.6789;
     String formattedUS = String.format(Locale.US, "US format: %, .2f", value);
     String formattedFR = String.format(Locale.FRANCE, "France format: %, .2f", value);
     System.out.println(formattedUS);
     System.out.println(formattedFR);
     // Example 2: Formatting strings
     String name = "Routh";
     int age = 25;
     String message = String.format(Locale.ENGLISH, "Hello %s, your age is %d.", name, age);
     System.out.println(message);
     // Example 3: Formatting with multiple data types
     String result = String.format(Locale.UK, "Name: %s, Score: %.1f, Passed: %b", "Alice",
95.5, true);
     System.out.println(result);
  }
}
```

## 5. What it Takes (Parameters)

- Locale 1 → Locale object specifying regional formatting rules (e.g., Locale.US, Locale.FRANCE).
- 2. String format → Format string containing placeholders:

```
○ %s → String
```

- o %d → Integer
- $\circ$  %f  $\rightarrow$  Floating-point number
- %b → Boolean
- o %n → Newline
- $\circ$  %, .2f  $\rightarrow$  Number with commas and 2 decimal points
- 3. Object... args  $\rightarrow$  Variable arguments corresponding to the placeholders.

## 6. Return Type

String → the formatted string according to the format and locale.

## 7. Printing Its Value

Already shown in the examples:

System.out.println(formattedUS); System.out.println(formattedFR); System.out.println(message); System.out.println(result);

#### **Sample Output:**

US format: 12,345.68

France format: 12 345,68 Hello Routh, your age is 25.

Name: Alice, Score: 95.5, Passed: true

If you want, I can make a **tiny table of all commonly used placeholders with examples** specifically for String.format(Locale, ...) so you can use it directly for problem solving.

Do you want me to do that?

# Stream<String> lines()

Got it! Let's go **step by step** for Stream<String> lines() in Java. This is commonly used with files or other input streams.

## 1. Syntax

Stream<String> lines = Files.lines(Path.of("filename.txt"));

 Files.lines(Path path) → returns a Stream<String> where each element is a line from the file.

#### If using **BufferedReader**:

BufferedReader reader = new BufferedReader(new FileReader("filename.txt")); Stream<String> lines = reader.lines();

#### 2. Purpose

- To read lines from a file or input stream lazily.
- Each line becomes an element of a **Stream**, which can be processed using **Stream** operations (map, filter, forEach, etc.).
- Useful for **large files**, because it doesn't read all lines into memory at once.

## 3. Priority in Problem Solving

- **High priority** when dealing with:
  - File I/O in Java.
  - Large text processing.
  - o Problems that require functional-style operations on lines.
- **Medium priority** in general algorithm problems unless input is from a file.

## 4. Core Examples / What it Takes

- Takes: nothing if using BufferedReader.lines(), or Path if using Files.lines(Path).
- Returns: Stream<String>.
- Can be used with Stream API methods.

#### Example 1: Reading a file line by line

```
import java.nio.file.*;
import java.io.IOException;
import java.util.stream.Stream;

public class Example {
    public static void main(String[] args) throws IOException {
        Stream<String> lines = Files.lines(Path.of("sample.txt"));
        lines.forEach(System.out::println); // prints each line
        lines.close(); // always close stream
    }
}
```

#### Example 2: Using BufferedReader

```
import java.io.*;
import java.util.stream.*;
```

## 5. Return Type

- Stream<String>
- Lazily reads the file lines; doesn't store all lines in memory at once.

## 6. Printing Values

• You can use forEach(System.out::println) to print each line:

lines.forEach(System.out::println);

• Or convert to a list if you want to store and print:

```
List<String> list = lines.toList();
list.forEach(System.out::println);
```

## Summary Table

```
Feature Details

Syntax Files.lines(Path) /
BufferedReader.lines()
```

Purpose Read file/input lines lazily as a Stream

Takes Path (Files) / nothing (BufferedReader)

Returns Stream<String>

Priority in problem High for file processing, medium otherwise

Core usage

example

Filtering, mapping, counting lines

Printing value lines.forEach(System.out::println

)

If you want, I can make a **single small program** that **reads a file, filters lines containing a word, and prints the result**—which is the most common use case in problem-solving.

Do you want me to do that?

## IntStream chars()

Sure! Let's break down IntStream chars() in Java step by step. I'll assume you mean the chars() method of the String class, since that's the commonly used one.

## 1. Syntax

IntStream chars()

- chars() is a method of the String class.
- It takes **no parameters**.

#### 2. Purpose

• To convert a **String** into an **IntStream** of **Unicode code points** (integers representing each character).

• It's mainly used for **functional-style operations** like filtering, mapping, or counting characters.

## 3. Priority to Use in Problem Solving

- **Medium to High** in string manipulation problems where:
  - You need character-level processing.
  - Counting vowels, digits, or certain characters.
  - Using streams and lambdas in modern Java.
- Not needed if just iterating with a for loop or charAt().
- Very handy in **competitive programming** or **functional programming style solutions**.

## 4. Core Examples / What it takes

- chars() takes nothing (()).
- Returns an IntStream of integer values corresponding to each character.

#### Example:

```
String str = "Hello";
IntStream stream = str.chars();
```

## 5. Return Type

java.util.stream.IntStream

• Each element is the **Unicode integer value** of a character.

## 6. Printing Its Value

You can print the values in different ways:

#### **Example 1: Print integer codes**

```
String str = "Hello";
str.chars().forEach(System.out::println);
```

#### **Output:**

72

101

108

108

111

#### **Example 2: Print characters**

```
String str = "Hello";
str.chars()
.forEach(c -> System.out.print((char)c + " "));
```

#### **Output:**

Hello

#### **Example 3: Using filtering in problem-solving**

#### **Output:**

Digits count: 3

## **✓** Summary Table

Feature Details

Class String

Method chars()

Parameters None

Return Type IntStream

**Purpose** Convert string to stream of Unicode integer codes for functional

processing

**Usage** Medium-High for functional string problems

**Priority** 

**Example** str.chars().forEach(c -> System.out.print((char)c));

If you want, I can also create a **small Java program showing chars() with multiple operations** like filtering vowels, digits, and converting to uppercase using IntStream. It will be a very handy reference for problem solving.

Do you want me to do that?

# IntStream codePoints()

Sure! Let's break down **IntStream codePoints()** from Java **String** class step by step in a structured way:

## 1 Syntax

IntStream codePoints()

- This method belongs to the String class.
- It does not take any parameters.

## 2 Purpose

- The codePoints() method returns a stream of Unicode code points from the string.
- Useful when you want to process characters as integers, especially for Unicode characters beyond the Basic Multilingual Plane (BMP).
- Helps in functional-style operations using streams.

## **3** Priority in Problem Solving

- Moderate priority in competitive programming or LeetCode-style problems.
- Mostly used when:
  - You need to manipulate or filter characters based on their Unicode values.
  - You want to sum character codes, map characters to integers, or process surrogate pairs safely.
- Not commonly used for simple string problems where regular char access suffices.

## 4 Core Examples (What It Takes and Returns)

```
Input: String str = "Hello";
```

• Returns: IntStream of Unicode code points of the string

```
String str = "Hello";
IntStream stream = str.codePoints(); // returns IntStream
```

```
    stream contains [72, 101, 108, 108, 111]
    (Unicode of 'H', 'e', 'l', 'l', 'o')
```

## **5** Return Type

java.util.stream.IntStream

• A **stream of integers** representing Unicode code points.

## 6 Printing Its Value

Since codePoints() returns a stream, you cannot print it directly. You need to **collect or iterate** it:

```
import java.util.stream.IntStream;

public class Main {
   public static void main(String[] args) {
        String str = "Hello";

        // Using forEach to print each code point
        System.out.print("Code Points: ");
        str.codePoints().forEach(cp -> System.out.print(cp + " "));
        System.out.println();

        // Or collect to an array
        int[] codePointsArray = str.codePoints().toArray();
        System.out.print("Code Points Array: ");
        for (int cp : codePointsArray) {
            System.out.print(cp + " ");
        }
    }
}
```

## **Output:**

Code Points: 72 101 108 108 111 Code Points Array: 72 101 108 108 111

If you want, I can also show a **real Unicode example** with emojis or non-BMP characters to demonstrate why codePoints() is better than using charAt() for advanced problem solving.

Do you want me to show that?