**1)Remove Outermost Parentheses**

**Example 1:**

Input: s = "((()))"

Output: "(())"

Explanation: The input string is a single primitive: "((()))".

Removing the outermost layer yields: "(())".

**Example 2:**

Input: s = "()(()())(())"

Output: "(()())()"

Explanation: Primitive decomposition: "()" + "(()())" + "(())"

After removing outermost parentheses: "" + "()()" + "()"

Final result: "(()())()".

**DryRun:**

1. Create one StringBuffer
2. Create the openCount variable
3. Use **forEach loop** inside loop convert string into array using **toCharArray()** method
4. Use 4 conditions

**class Solution {**

**public String removeOuterParentheses(String s) {**

**StringBuilder result = new StringBuilder();**

**int openCount = 0;**

**for (char c : s.toCharArray()) {**

**if (c == '(') {**

**if (openCount > 0) {**

**result.append(c);**

**}**

**openCount++;**

**} else if (c == ')') {**

**openCount--;**

**if (openCount > 0) {**

**result.append(c);**

**}**

**}**

**}**

**return result.toString();**

**}**

**}**

**Time Complexity: O(n)**, since we are performing a single traversal of the string.  
**Space Complexity: O(1)**, since we are using a few variables to track the current state.

2)**Reverse Words in a String**

**Approach**

* Use a stack to push all the words in a stack
* Now, all the words of the string are present in the stack, but in reverse order
* Pop elements of the stack one by one and add them to our answer variable. Remember to add a space between the words as well.
* Here’s a quick demonstration of the same

**🧩 Summary of How It Works**

1. Adds a space at the end of the string.
2. Reads characters one by one. By creating 1 variable
3. Whenever a space is found → pushes the word to the stack.
4. After reading all words, pops each word from the stack to reverse order.
5. Prints the reversed sentence.

**CODE:**

**import java.util.\*;**

**class Test**

**{**

**public static void main(String[] args)**

**{**

**String s = "TUF is great for interview preparation";**

**System.out.println("After reversing words: ");**

**System.out.println(s);**

**s += " ";**

**Stack<String> st = new Stack<String>();**

**int i;**

**String str = "";**

**for (i = 0;i < s.length();i++)**

**{**

**if (s.charAt(i) == ' ')**

**{**

**st.push(str);**

**str = "";**

**}**

**else**

**{**

**str += s.charAt(i);**

**}**

**}**

**String ans = "";**

**while (st.size() != 1)**

**{**

**ans += st.peek() + " ";**

**st.pop();**

**}**

**ans += st.peek(); *// The last word should'nt have a space after it***

**System.out.println("After reversing words: ");**

**System.out.print(ans);**

**}}**

**Approach:**

* We start traversing the string from the end until we hit a space. It indicates that we have gone past a word and now we need to store it.
* We check if our answer variable is empty or not
* If it’s empty, it indicates that this is the last word we need to print, and hence, there shouldn’t be any space after this word.
* If it’s empty we add it to our result with a space after it. Here’s a quick demonstration of the same

**NOTE**:

**s.split():**

**What is split()?**

* The split() method in Java **divides a string into parts (substrings)** based on a **delimiter** (a separator like space, comma, etc.).
* It returns an **array of strings**.

**Example:**String s = "Java is fun";

String[] arr = s.split(" ");

**Output**:arr = ["Java", "is", "fun"]

**NOTE**:

**What does "\\s+" mean?**

This is a **regular expression (regex)** pattern used inside split().

Let’s decode it:

| **Symbol** | **Meaning** |
| --- | --- |
| **\s->** | Matches any **whitespace** character (like space, tab, or newline) |
| **+->** | Means **“one or more”** occurrences |

* class Solution {
* public String reverseWords(String s) {
* s = s.trim();
* String[] words = s.split("\\s+");
* StringBuilder sb = new StringBuilder();
* for(int i = words.length - 1; i >= 0; i--){
* sb.append(words[i]);
* if(i > 0){
* sb.append(" ");
* }
* }
* return sb.toString();
* }
* }

**Time Complexity: O(N),** N~length of string

**Space Complexity: O(1),** Constant Space

**NOTE:**

**🧩 1️⃣ split() — breaks a string into words or parts**

**Definition:**

The split() method divides a string into **multiple substrings** based on a **delimiter (like space, comma, etc.)** and returns a **String array (String[])**.

**Syntax:**

String[] arr = s.split(" ");

**Example:**

String s = "Java is fun";

String[] words = s.split(" ");

for (String w : words) {

System.out.println(w);

}

**Output:**

Java

is

fun

🧠 **Explanation:**

* split(" ") → splits whenever it finds a **space**.
* Returns **String array** (each element is a word).
* split() is used when we want to **work with words**, not individual characters.

**🧩 2️⃣ toCharArray() — breaks a string into characters**

**Definition:**

It converts the entire string into a **character array (char[])**.

**Example:**

String s = "Java";

char[] ch = s.toCharArray();

for (char c : ch) {

System.out.println(c);

}

**Output:**

J

a

v

a

🧠 **Explanation:**

* Breaks **every character**, including spaces, into array elements.
* Used when we need to **traverse or modify characters** (like reversing).

**🧩 3️⃣ charAt(index) — gets one character at a time**

**Definition:**

Returns the character present at the given index.

**Example:**

String s = "Java";

System.out.println(s.charAt(2));

**Output:**

v

🧠 **Explanation:**

* Used when you want **only one character**.
* No new array is created — it just accesses one position directly.

**🧠 Important Note about charAt()**

* The **charAt()** method **always requires an index value** (an integer).
* This index tells **which character** in the string you want to access.
* Indexing in Java strings starts from **0** (zero-based indexing).

3)**Largest Odd Number in a String.**

class Solution {

    public String largestOddNumber(String num) {

        for (int i = num.length() - 1; i >= 0; i--) {

            int digit = num.charAt(i) - '0';  // Convert char to int

            if (digit % 2 != 0) {

                return num.substring(0, i + 1); // Substring up to last odd digit

            }

        }

        return ""; // No odd digit found}}

| **Expression** | **Start** | **End** | **Includes Indexes** | **Output** |
| --- | --- | --- | --- | --- |
| substring(0, 3) | 0 | 3 | 0, 1, 2 | "420" |
| substring(0, 4) | 0 | 4 | 0, 1, 2, 3 | "4205" |
| substring(0, 5) | 0 | 5 | 0, 1, 2, 3, 4 | "42053" |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4)**Longest Common Prefix**  **Example 1**  **Input:** str = ["flower", "flow", "flight"]  **Output:** "fl"  **Explanation:** All strings in the array begin with the common prefix "fl".  **Example 2**  **Input:** str = ["apple", "banana", "grape", "mango"]  **Output:** ""  **Explanation:** None of the strings share a common starting sequence, so the result is an empty string.  **NOTE:**  The **substring() method in Java** is used to extract a part of a string. It returns a new string that starts from the given startIndex and ends **before** the given end  Index.  **CODE:** |  |  |  |  |

class Solution {

    public String longestCommonPrefix(String[] strs) {

        if (strs == null || strs.length == 0) return "";

        String prefix = strs[0];  // Start with the first word as prefix

        for (int i = 1; i < strs.length; i++) {

            // Keep reducing prefix until the word starts with it

            while (!strs[i].startsWith(prefix)) {

                prefix = prefix.substring(0, prefix.length() - 1);

                if (prefix.isEmpty()) {

                    return "";

                }

            }

        }

        return prefix;

    }

}

**Time Complexity**

* n = number of strings in the array (strs.length)
* m = length of the first string (strs[0].length()) — assume the maximum length of any string is “**m**”

✅ **Time Complexity:** O(n \* m²) in the worst case

In the worst case, we **compare each character of the prefix**.

Prefix can be at most length m → O(m) comparisons per word.

**5)Isomorphic String**

**Problem Statement:**Given two strings s and t, determine if they are isomorphic. Two strings s and t are isomorphic if the characters in s can be replaced to get t.  
All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

**Examples**

**Example 1**

**Input:** s = "paper", t = "title"

**Output:** true

**Explanation:** The characters in "s" can be mapped one-to-one to characters in "t":

'p' → 't', 'a' → 'i', 'e' → 'l', 'r' → 'e'

Since the mapping is consistent and unique for each character, the strings are isomorphic.

**Example 2**

**Input:** s = "foo", t = "bar"

**Output:** false

**Explanation:** 'f' → 'b' is fine, 'o' → 'a' for the first 'o', But the second 'o' in "s" would need to map to 'r' in "t", which conflicts with the earlier mapping of 'o' → 'a'

This inconsistency makes it impossible to convert "s" to "t" using a one-to-one character mapping.

class Solution {

    public boolean isIsomorphic(String s, String t) {

        if (s.length() != t.length()) return false;

// In Java, a char is based on Unicode (which can represent far more than 256 characters).

// But in many algorithmic problems like this one (LeetCode, etc.), the input is restricted to ASCII characters.

// ASCII has 256 possible values (0–255);

        int[] mapS = new int[256]; // mapping from s → t

        int[] mapT = new int[256]; // mapping from t → s

        for (int i = 0; i < s.length(); i++) {

            char ch1 = s.charAt(i);

            char ch2 = t.charAt(i);

            if (mapS[ch1] == 0 && mapT[ch2] == 0) {

                // First time mapping

                mapS[ch1] = ch2;

                mapT[ch2] = ch1;

            } else {

                // Already mapped, check consistency

                if (mapS[ch1] != ch2 || mapT[ch2] != ch1) {

                    return false;

                }

            }

        }

        return true;

    }

}

Two strings are **isomorphic** if each character in the first string maps to **exactly one character** in the second string, and vice versa.

* "egg" → "add" works because:
  + 'e' always maps to 'a'
  + 'g' always maps to 'd'
  + No conflicts.
* "foo" → "bar" fails because:
  + 'o' in "foo" maps to 'a' the first time, but the next 'o' tries to map to 'r'.
  + This violates the rule of one-to-one mapping → not isomorphic.

**🔹 Key words to remember:**

* “One-to-one mapping”
* “Each character must map consistently”
* “No conflicts allowed”