Java Code Solutions to Top 20 String Problems

1. Longest substring without repeating characters

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}           set.add(s.charAt(right));           maxLen = Math.max(maxLen, right - left + 1);
}     return maxLen; }
```

2. Longest palindromic substring

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}           set.add(s.charAt(right));           maxLen = Math.max(maxLen, right - left + 1);
}     return maxLen; }
```

3. Check if two strings are anagrams

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}           set.add(s.charAt(right));           maxLen = Math.max(maxLen, right - left + 1);
}     return maxLen; }
```

4. Valid palindrome check

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}           set.add(s.charAt(right));           maxLen = Math.max(maxLen, right - left + 1);
}     return maxLen; }
```

5. String compression (e.g., aabcccccaaa -> a2b1c5a3)

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
```

```
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

6. Count and say sequence

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

7. Implement string to integer (atoi)

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

8. Implement strstr() / indexOf()

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

9. Group anagrams

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

10. Reverse words in a string

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
```

```
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

11. Check if two strings are isomorphic

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

12. Longest common prefix

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

13. Minimum window substring

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

14. Permutation in string

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

15. Count palindromic substrings

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
```

```
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

16. All unique characters in a string

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

17. Remove duplicate characters

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

18. First non-repeating character

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

19. Generate all permutations of a string

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}          set.add(s.charAt(right));          maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

20. Convert Roman numeral to integer

```
public int lengthOfLongestSubstring(String s) {       Set<Character> set = new
HashSet<>();
```

```
int left = 0, maxLen = 0;     for (int right = 0; right < s.length(); right++) {
while   (set.contains(s.charAt(right)))   {
set.remove(s.charAt(left++));
}           set.add(s.charAt(right));           maxLen = Math.max(maxLen, right -
left + 1);
}     return maxLen; }
```

Java Code Solutions to Interview Problems

21. 1. Two sum problem

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

22. 2. Three sum problem

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

23. 3. Move zeroes to the end

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

24. 4. Find duplicate number without modifying array

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

25. 5. Merge two sorted arrays

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

26. 6. Kadane's algorithm for max subarray sum

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

27. 7. Find the majority element

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

28. 8. Longest consecutive sequence

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

29. 9. Product of array except self

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

30. 10. Next permutation of numbers

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

31. 11. Subarray sum equals k

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

32. 12. Longest substring with at most k distinct characters

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

33. 13. Top K frequent elements

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

34. 14. Word pattern match

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

35. 15. Design an LRU cache

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                        return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

36. 16. Valid parentheses

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                         return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

37. 17. Min stack with getMin in O(1)

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                         return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

38. 18. Evaluate reverse polish notation

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                         return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

39. 19. Implement queue using stacks / stack using queues

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                         return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

40. 20. N-Queens problem

```
public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map =
new HashMap<>();
for (int i = 0; i < nums.length; i++) {            int complement = target -
nums[i];         if
(map.containsKey(complement))  {                         return  new  int[]  {
map.get(complement), i };         }
map.put(nums[i], i);     }     return new int[0]; }
```

Java Code Solutions to Interview Problems

41. 1. Sudoku solver

```
public class SudokuSolver {     public void solveSudoku(char[][] board) {
 solve(board);
}      private boolean solve(char[][] board) {          for (int i = 0; i < 9;
i++) {             for
(int j = 0; j < 9; j++) {                if (board[i][j] == '.') {
       for (char c
= '1'; c <= '9'; c++) {                       if (isValid(board, i, j, c)) {
board[i][j] = c;                           if (solve(board)) return true;
board[i][j] = '.';                          }                    }
      return false;
}               }          }           return true;      }      private boolean
isValid(char[][] board, int
row, int col, char c) {          for (int i = 0; i < 9; i++) {              if
(board[i][col] == c ||
board[row][i] == c ||                board[3 * (row / 3) + i / 3][3 * (col / 3)
+ i % 3] == c)
return false;         }         return true;     } }
```

42. 2. Letter combinations of a phone number

```
public List<String> letterCombinations(String digits) {      if (digits == null
|| digits.length() ==
0) return new ArrayList<>();       String[] mapping = {"0", "1", "abc", "def",
"ghi", "jkl", "mno",
"pqrs", "tuv", "wxyz"};        List<String> result = new ArrayList<>();
backtrack(result, digits,
"", 0, mapping);        return result; } private void backtrack(List<String>
result, String digits,
String current, int index, String[] mapping) {     if (index == digits.length())
{
result.add(current);                 return;         }         String letters =
mapping[digits.charAt(index) - '0'];
for (char c : letters.toCharArray()) {         backtrack(result, digits, current
+ c, index + 1,
mapping);     } }
```

43. 3. Word break problem

```
public boolean wordBreak(String s, List<String> wordDict) {       Set<String>
wordSet = new
HashSet<>(wordDict);     boolean[] dp = new boolean[s.length() + 1];     dp[0] =
true;     for (int
i = 1; i <= s.length(); i++) {         for (int j = 0; j < i; j++) {
if (dp[j] &&
wordSet.contains(s.substring(j, i))) {                dp[i] = true;
    break;
}         }     }     return dp[s.length()]; }
```

44. 4. Subsets and combinations of array

```
public List<List<Integer>> subsets(int[] nums) {      List<List<Integer>> result
= new ArrayList<>();
backtrack(0, nums, new ArrayList<>(), result);     return result; } private void
backtrack(int
start, int[] nums, List<Integer> temp, List<List<Integer>> result) {
result.add(new
ArrayList<>(temp));      for (int i = start; i < nums.length; i++) {
temp.add(nums[i]);
backtrack(i + 1, nums, temp, result);          temp.remove(temp.size() - 1);
} }
```

45. 5. Find all binary strings without consecutive 1s

```
public void generateBinaryStrings(int n, String str, int lastDigit) {     if (n
== 0) {
System.out.println(str);          return;     }     generateBinaryStrings(n - 1,
str + "0", 0);
if (lastDigit == 0)         generateBinaryStrings(n - 1, str + "1", 1); }
```

46. 6. Lowest common ancestor in binary tree

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
 if (root == null
||  root  ==  p  ||  root  ==  q)  return  root;           TreeNode  left  =
lowestCommonAncestor(root.left, p, q);
TreeNode right = lowestCommonAncestor(root.right, p, q);     return left == null
? right : right ==
null ? left : root; }
```

47. 7. Serialize and deserialize binary tree

```
public class Codec {      public String serialize(TreeNode root) {          if
(root == null) return
"null,";                   return  root.val  +  ","  +  serialize(root.left)  +
serialize(root.right);     }
public TreeNode deserialize(String data) {         Queue<String> nodes = new
LinkedList<>(Arrays.asList(data.split(",")));          return buildTree(nodes);
  }     private
TreeNode buildTree(Queue<String> nodes) {         String val = nodes.poll();
    if
("null".equals(val))  return  null;                    TreeNode  node  =  new
TreeNode(Integer.parseInt(val));
node.left = buildTree(nodes);          node.right = buildTree(nodes);
return node;     } }
```

48. 8. Detect cycle in graph

```
public boolean hasCycle(int V, List<List<Integer>> adj) {     boolean[] visited
= new boolean[V];
```

```
boolean[] recStack = new boolean[V];     for (int i = 0; i < V; i++) {
if (dfs(i, visited,
recStack, adj)) return true;     }     return false; } private boolean dfs(int
v, boolean[] visited,
boolean[] recStack, List<List<Integer>> adj) {     if (recStack[v]) return true;
    if (visited[v])
return false;       visited[v] = true;       recStack[v] = true;       for (int
neighbor : adj.get(v)) {
if (dfs(neighbor, visited, recStack, adj)) return true;     }     recStack[v] =
false;     return
false; }
```

49. 9. Clone a graph

```
public Node cloneGraph(Node node) {       if (node == null) return null;
Map<Node, Node> map = new
HashMap<>();     Queue<Node> queue = new LinkedList<>();     queue.add(node);
 map.put(node, new
Node(node.val));         while (!queue.isEmpty()) {               Node curr =
queue.poll();         for (Node
neighbor : curr.neighbors) {             if (!map.containsKey(neighbor)) {
map.put(neighbor, new Node(neighbor.val));                 queue.add(neighbor);
          }
map.get(curr).neighbors.add(map.get(neighbor));           }       }       return
map.get(node); }
```

50. 10. Level order traversal of binary tree

```
public List<List<Integer>> levelOrder(TreeNode root) {       List<List<Integer>>
result = new
ArrayList<>();     if (root == null) return result;     Queue<TreeNode> queue =
new LinkedList<>();
queue.add(root);     while (!queue.isEmpty()) {         int size = queue.size();
List<Integer> level = new ArrayList<>();         for (int i = 0; i < size; i++)
{
TreeNode node = queue.poll();             level.add(node.val);             if
(node.left != null)
queue.add(node.left);             if (node.right != null) queue.add(node.right);
        }
result.add(level);     }     return result; }
```

51. 11. Check if binary tree is symmetric

```
public boolean isSymmetric(TreeNode root) {     if (root == null) return true;
  return
isMirror(root.left,  root.right);  }  private  boolean  isMirror(TreeNode  t1,
TreeNode t2) {     if (t1
== null && t2 == null) return true;      if (t1 == null || t2 == null) return
false;     return
(t1.val == t2.val) &&             isMirror(t1.left, t2.right) &&
isMirror(t1.right,
```

```
t2.left); }
```

52. 12. Validate a binary search tree

```
public boolean isValidBST(TreeNode root) {        return validate(root, null, null); } private boolean
validate(TreeNode node, Integer low, Integer high) {       if (node == null) return true;     if ((low
!= null && node.val <= low) || (high != null && node.val >= high))
return false;     return
validate(node.left, low, node.val) && validate(node.right, node.val, high); }
```

53. 13. Immutable class implementation

```
public final class Person {     private final String name;     private final int age;     public
Person(String name, int age) {         this.name = name;         this.age = age;     }     public
String getName() { return name; }     public int getAge() { return age; } }
```

54. 14. Override equals() and hashCode()

```
@Override public boolean equals(Object o) {     if (this == o) return true;
if (o == null ||
getClass() != o.getClass()) return false;      Person person = (Person) o;
return age ==
person.age && name.equals(person.name); } @Override public int hashCode() {
return
Objects.hash(name, age); }
```

55. 15. Implement Comparator for sorting

```
public class AgeComparator implements Comparator<Person> {        public int compare(Person a, Person
b) {         return Integer.compare(a.getAge(), b.getAge());     } }
```

56. 16. Difference between HashMap, LinkedHashMap, TreeMap

```
Map<String, Integer> map1 = new HashMap<>(); Map<String, Integer> map2 = new LinkedHashMap<>();
Map<String, Integer> map3 = new TreeMap<>(); map1.put("b", 2); map1.put("a", 1); map2.put("b", 2);
map2.put("a",       1);       map3.put("b",       2);       map3.put("a",       1);
System.out.println("HashMap: " + map1);
System.out.println("LinkedHashMap: " + map2); System.out.println("TreeMap: " + map3);
```

57. 17. Thread-safe Singleton pattern

```
public class Singleton {      private static volatile Singleton instance;
private Singleton() {}
```

```
public static Singleton getInstance() {         if (instance == null) {
   synchronized
(Singleton.class) {                       if (instance == null) instance = new
Singleton();             }
}         return instance;     } }
```

58. 18. Producer-consumer using BlockingQueue

```
BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(5); Runnable producer =
() -> {     try {
int value = 0;         while (true) {             queue.put(value);
System.out.println("Produced: " + value++);             Thread.sleep(1000);
   }     } catch
(InterruptedException e) { e.printStackTrace(); } }; Runnable consumer = () -> {
    try {
while (true) {                     int val = queue.take();
System.out.println("Consumed: " +
val);                       Thread.sleep(1500);               }         } catch
(InterruptedException e) {
e.printStackTrace();     }     };     new     Thread(producer).start();     new
Thread(consumer).start();
```

59. 19. Valid parentheses

```
public boolean isValid(String s) {     Stack<Character> stack = new Stack<>();
  for (char c :
s.toCharArray()) {         if (c == '(') stack.push(')');         else if (c ==
'{')
stack.push('}');          else if (c == '[') stack.push(']');          else if
(stack.isEmpty() ||
stack.pop() != c) return false;     }     return stack.isEmpty(); }
```

60. 20. Min stack with getMin in O(1)

```
class MinStack {      Stack<Integer> stack = new Stack<>();      Stack<Integer>
minStack = new
Stack<>();     public void push(int val) {          stack.push(val);          if
(minStack.isEmpty()
|| val <= minStack.peek()) minStack.push(val);     }     public void pop() {
    if
(stack.pop().equals(minStack.peek())) minStack.pop();     }     public int top()
{ return
stack.peek(); }     public int getMin() { return minStack.peek(); } }
```