# ⭐ LEVEL 1 (60 QUESTIONS — Very Basic)

1. Print “Hello World”.
2. Take two numbers and print their sum.
3. Take two numbers and print their difference.
4. Take two numbers and print their product.
5. Take two numbers and print their division result.
6. Swap two variables using a third variable.
7. Swap two variables without using a third variable.
8. Check if a number is even or odd.
9. Check if a number is positive, negative, or zero.
10. Check if a year is leap year or not.
11. Check if a number is integer or decimal.
12. Find square root of a number (Math.sqrt).
13. Find power of a number (Math.pow).
14. Print multiplication table of N.
15. Print numbers from 1 to N.
16. Print all even numbers from 1 to N.
17. Print all odd numbers from 1 to N.
18. Find sum of numbers from 1 to N.
19. Find sum of squares from 1 to N.
20. Find sum of cubes from 1 to N.
21. Find sum of factors of a number.
22. Count digits in a number.
23. Reverse a number.
24. Check if a number is palindrome.
25. Find sum of digits of a number.
26. Product of digits of a number.
27. Armstrong number check (3 digit).
28. Find maximum of 3 numbers.
29. Find minimum of 3 numbers.
30. Check if a character is vowel or consonant.
31. Find ASCII value of a character.
32. Print first N Fibonacci numbers.
33. Check if a number belongs to Fibonacci sequence.
34. Check if a number is perfect.
35. Check if a number is sunny.
36. Check if a number is happy.
37. Check if a number is spy (sum of digits == product of digits).
38. Check if a number is prime.
39. Print all primes in a range.
40. Find GCD of two numbers (iterative).
41. Find LCM of two numbers.
42. Print factorial of a number (loop).
43. Take array input (space-separated).

44. Sum of array elements.
45. Max element of array.
46. Min element of array.
47. Average of array elements.
48. Count even and odd numbers in array.
49. Find absolute difference between sum of even and sum of odd elements.
50. Linear search in array.
51. Reverse an array.
52. Reverse a string.
53. Count vowels in a string.
54. Count consonants in a string.
55. Convert string to uppercase.
56. Convert string to lowercase.
57. Find frequency of a character.
58. Check if string is palindrome.
59. Remove spaces from string.
60. Remove digits from a string.

---

# ⭐ LEVEL 2 (50 QUESTIONS — Beginner → Solid Foundation)

61. Remove special characters from a string.
62. Count special characters in a string.
63. Convert string digits into integer sum ("a1b2c3" → 1+2+3).
64. Reverse words in a string ("I love you" → "you love I").
65. Reverse each word in string ("I love you" → "I evol uoy").
66. Find sum of ASCII values in string.
67. Convert "abcd" → 1+2+3+4.
68. Convert "abcd" → "zyxw".
69. Replace vowels with next vowel.
70. Expand encoded string ("g3v2k5" → gggvvkkkkk).
71. Extract largest number from string ("ga56v123k9" → 123).
72. Check if two strings are anagrams.
73. Remove duplicates from sorted array.
74. Remove duplicates from unsorted array.
75. Check if array is subset of another.
76. Binary search on sorted array.
77. Selection sort.
78. Bubble sort.
79. Count prime numbers in array.
80. Sum of prime numbers in array.
81. Second largest element in array.

82. Second smallest element in array.
83. Find pair sum == K (unordered array).
84. Find pair sum == K (sorted array – two pointer).
85. Find pair whose product == K (sorted array).
86. Rotate array by K (left rotation).
87. Rotate array by K (right rotation).
88. Print all substrings of a string.
89. Print frequency of each element in array (HashMap).
90. Print duplicates in array (HashMap).
91. Remove duplicates in string (HashSet).
92. Count word frequency in a sentence.
93. Count character frequency in a string (HashMap).
94. Print elements by increasing frequency.
95. Print elements by decreasing frequency.
96. Largest element in matrix.
97. Count zeros in matrix.
98. Check if matrix is diagonal matrix.
99. Sum of primary diagonal.
100. Sum of secondary diagonal.

---

# ⭐ LEVEL 3 (40 QUESTIONS — Strong Beginner)

101. Check if number is automorphic.
102. Check if number is harshad.
103. Check if number is abundant.
104. Print prime factors of a number.
105. Find maximum digit in a number.
106. Find minimum digit in a number.
107. Replace all zeros with ones in number.
108. Express number as sum of two primes.
109. Find median of array.
110. Count distinct elements in array.
111. Move all zeros to end.
112. Move all negatives to one side.
113. Find missing number in 1–N array.
114. Find duplicate element (single duplicate).
115. Find all duplicates.
116. Find first non-repeating character in string.
117. Find first repeating character.
118. Remove vowels from string.
119. Sort characters in a string.

120. Sort string by frequency.
121. Largest word in string.
122. Word with highest repeating characters.
123. Count number of words in string.
124. Replace each character with next lexicographic alphabet.
125. Sum of numbers in string ("a12b3" → 12+3).
126. Convert words to numbers ("one two three").
127. Convert numbers to words (1–9999).
128. Insertion sort (simple).
129. Kadane's algorithm (max subarray sum) – easy version.
130. Count subarrays with sum zero.
131. Count pairs whose sum = K (HashMap).
132. Basic prefix sum array.
133. Basic suffix sum array.
134. Find equilibrium index.
135. Check if two strings match with wildcard (* and ?).
136. Check if number is strong number.
137. Sum of GP series.
138. Sum of AP series.
139. Add two fractions.
140. Permutations: N people → R seats.

---

# ⭐ LEVEL 4 (30 QUESTIONS — Pre-Intermediate Foundation)

141. Next greater element (naive).
142. Next smaller element (naive).
143. Sort array by frequency (HashMap + sorting).
144. Find top 2 frequent elements.
145. Find elements occurring more than N/3 times.
146. Longest word in sentence.
147. Print all permutations of string (recursion).
148. Print all subsets of string (recursion).
149. Binary exponentiation (power using logN).
150. Check if two arrays are equal (using HashMap).
151. Count number of anagram pairs in list of strings.
152. Left rotate string by K.
153. Right rotate string by K.
154. Longest substring without repeating characters (easy version).
155. Check if a matrix is symmetric.
156. Transpose of matrix.
157. Rotate matrix 90 degrees.

158. Spiral printing of matrix.
159. Binary to decimal.
160. Decimal to binary.
161. Binary to octal.
162. Octal to decimal.
163. Decimal to octal.
164. Octal to binary.
165. Check balanced parentheses (stack logic, but simple).
166. Remove brackets from algebraic expression.
167. Sum of primary + secondary diagonal difference.
168. Print Z pattern of matrix.
169. Print boundary elements of matrix.
170. Reverse words in place in char array.

---

# ⭐ LEVEL 5 (20 QUESTIONS — Intermediate Foundation / Pre-DSA)

171. Sliding window: first window sum size K.
172. Sliding window: max sum subarray of size K.
173. Sliding window: count occurrences of anagram (easy version).
174. Two pointer: count pairs with given difference.
175. Two pointer: remove duplicates from sorted array (in-place).
176. HashMap: longest subarray with given sum K.
177. HashMap: longest subarray with equal 0s and 1s.
178. String: longest palindrome (bruteforce).
179. String: check rotation (“abcde”, “cdeab”).
180. String: minimum character deletions to make anagram.
181. Array: stock buy-sell to get max profit (1 transaction).
182. Array: rainwater trapping (easy version).
183. Array: find all triplets with sum = K.
184. Array: find product of array except self (no division).
185. Recursion: power set generation.
186. Recursion: tower of Hanoi (3 rods).
187. Recursion: sum of array.
188. Recursion: check palindrome string.
189. Recursion: binary search.
190. Recursion: Fibonacci optimized using memoization.
191. Check if string is valid shuffle of two strings.
192. Count distinct substrings (basic).
193. Print all palindromic substrings.
194. Check if two strings are isomorphic.
195. First negative number in every window of size K.

196. Check if array can be rearranged to form consecutive sequence.
197. Find frequency of each word in paragraph (HashMap).
198. Sort array based on absolute difference from a value X.
199. Rearrange array in wave form.
200. Longest consecutive number sequence (HashSet).

# 1) ArrayList — 25 Questions

1. **Create an ArrayList of integers and add 5 numbers.**
   Hint: Use `ArrayList<Integer>` and `add()` method.
   Expected: List contains 5 numbers in insertion order.
2. **Print all elements of the ArrayList using for-each loop.**
   Hint: Use enhanced for-loop.
   Expected: Numbers printed line by line.
3. **Print all elements using a standard for loop (with index).**
   Hint: Use `get(index)` method.
   Expected: Numbers printed in order with their indices.
4. **Remove the 3rd element from the ArrayList.**
   Hint: Use `remove(index)`.
   Expected: List has 4 elements; 3rd removed.
5. **Remove a specific value (e.g., 10) from ArrayList.**
   Hint: Use `remove(Object o)`.
   Expected: If 10 exists, it's removed.
6. **Check if ArrayList contains a value (e.g., 20).**
   Hint: Use `contains()`.
   Expected: `true` or `false`.
7. **Find the size of the ArrayList.**
   Hint: Use `size()`.
   Expected: Integer representing number of elements.
8. **Insert a number at 2nd index.**
   Hint: Use `add(index, value)`.
   Expected: Number appears at 2nd index; others shift right.
9. **Update element at 1st index to a new value.**
   Hint: Use `set(index, value)`.
   Expected: 1st element replaced.

10. **Clear all elements from ArrayList.**
    Hint: Use `clear()`.
    Expected: List becomes empty.
11. **Check if ArrayList is empty.**
    Hint: Use `isEmpty()`.
    Expected: `true` if list has no elements.
12. **Find the index of a number (e.g., 50) in ArrayList.**
    Hint: Use `indexOf()`.
    Expected: Returns index or -1.
13. **Find the last index of a number (duplicate case).**
    Hint: Use `lastIndexOf()`.
    Expected: Returns last occurrence index.
14. **Convert ArrayList to array.**
    Hint: Use `toArray()`.
    Expected: Elements copied into array.
15. **Sort the ArrayList in ascending order.**
    Hint: Use `Collections.sort(list)`.
    Expected: Elements in ascending order.
16. **Sort the ArrayList in descending order.**
    Hint: Use `Collections.sort(list, Collections.reverseOrder())`.
    Expected: Elements in descending order.
17. **Reverse the ArrayList.**
    Hint: Use `Collections.reverse(list)`.
    Expected: List elements appear in reverse.
18. **Find max element in ArrayList.**
    Hint: Use `Collections.max(list)`.
    Expected: Returns maximum number.
19. **Find min element in ArrayList.**
    Hint: Use `Collections.min(list)`.
    Expected: Returns minimum number.
20. **Sum all elements of ArrayList.**
    Hint: Iterate and add numbers.
    Expected: Integer sum of all elements.
21. **Remove duplicates from ArrayList.**
    Hint: Use a `HashSet` temporarily.
    Expected: List has only unique elements.
22. **Print only even numbers from ArrayList.**
    Hint: Use modulus `%` check.
    Expected: Only even numbers printed.
23. **Print only odd numbers from ArrayList.**
    Hint: Use modulus `%` check.
    Expected: Only odd numbers printed.
24. **Find average of elements in ArrayList.**
    Hint: Sum all elements and divide by `size()`.
    Expected: Floating-point average.

25. **Merge two ArrayLists into one.**
    Hint: Use `addAll()`.
    Expected: Single list with all elements from both lists.

# 2) LinkedList — 20 Questions

1. **Create a LinkedList of integers and add 5 numbers.**
   Hint: Use `LinkedList<Integer>` and `add()`.
   Expected: List contains 5 numbers in insertion order.
2. **Print all elements using for-each loop.**
   Hint: Use enhanced for-loop.
   Expected: Numbers printed in order.
3. **Print all elements using standard for loop with `get(index)`.**
   Hint: Use `get(index)` method.
   Expected: Numbers printed with indices.
4. **Add element at the beginning of LinkedList.**
   Hint: Use `addFirst(value)`.
   Expected: Element becomes head of list.
5. **Add element at the end of LinkedList.**
   Hint: Use `addLast(value)` or `add(value)`.
   Expected: Element appended at tail.
6. **Remove first element of LinkedList.**
   Hint: Use `removeFirst()`.
   Expected: Head removed; size decreases by 1.
7. **Remove last element of LinkedList.**
   Hint: Use `removeLast()`.
   Expected: Tail removed.
8. **Check if LinkedList contains a value (e.g., 10).**
   Hint: Use `contains()`.
   Expected: `true` or `false`.
9. **Find size of LinkedList.**
   Hint: Use `size()`.
   Expected: Returns number of elements.
10. **Update element at 2nd index.**
    Hint: Use `set(index, value)`.
    Expected: Element replaced.
11. **Insert element at 3rd index.**
    Hint: Use `add(index, value)`.
    Expected: Element inserted; others shift.
12. **Remove element at 4th index.**
    Hint: Use `remove(index)`.
    Expected: Element deleted; list shifts.
13. **Iterate using Iterator.**
    Hint: Use `Iterator<Integer>` and `hasNext()`.
    Expected: Elements printed in order.

14. **Iterate using descending iterator.**
    Hint: Use `descendingIterator()`.
    Expected: Elements printed in reverse.
15. **Convert LinkedList to ArrayList.**
    Hint: Use constructor `new ArrayList<>(linkedList)`.
    Expected: New ArrayList contains same elements.
16. **Sort LinkedList in ascending order.**
    Hint: Use `Collections.sort(linkedList)`.
    Expected: Elements sorted.
17. **Sort LinkedList in descending order.**
    Hint: Use `Collections.sort(linkedList, Collections.reverseOrder())`.
    Expected: Elements sorted descending.
18. **Remove duplicates from LinkedList.**
    Hint: Use `HashSet` temporarily.
    Expected: Only unique elements remain.
19. **Find max element in LinkedList.**
    Hint: Use `Collections.max(linkedList)`.
    Expected: Returns maximum value.
20. **Find min element in LinkedList.**
    Hint: Use `Collections.min(linkedList)`.
    Expected: Returns minimum value.

# 3) HashSet & LinkedHashSet — 20 Questions

1. **Create a HashSet of integers and add 5 numbers.**
   Hint: Use `HashSet<Integer>` and `add()`.
   Expected: Set contains 5 unique numbers (order not guaranteed).
2. **Print all elements of HashSet.**
   Hint: Use for-each loop.
   Expected: All elements printed (order arbitrary).
3. **Check if HashSet contains a number (e.g., 10).**
   Hint: Use `contains()`.
   Expected: `true` or `false`.
4. **Remove a number from HashSet.**
   Hint: Use `remove()`.
   Expected: Element deleted if present.
5. **Find size of HashSet.**
   Hint: Use `size()`.
   Expected: Integer count of unique elements.
6. **Add duplicate elements to HashSet.**
   Hint: Add same value twice.
   Expected: Only one copy stored.

7. **Clear all elements from HashSet.**
   Hint: Use `clear()`.
   Expected: Set becomes empty.
8. **Check if HashSet is empty.**
   Hint: Use `isEmpty()`.
   Expected: `true` if no elements.
9. **Iterate HashSet using Iterator.**
   Hint: Use `Iterator<Integer>` and `hasNext()`.
   Expected: All elements printed.
10. **Convert HashSet to ArrayList.**
    Hint: Use `new ArrayList<>(hashSet)`.
    Expected: ArrayList contains same elements.
11. **Create LinkedHashSet and add 5 numbers.**
    Hint: Use `LinkedHashSet<Integer>`.
    Expected: Elements preserve insertion order.
12. **Print all elements of LinkedHashSet.**
    Hint: Use for-each loop.
    Expected: Elements printed in insertion order.
13. **Remove duplicates from a list using LinkedHashSet.**
    Hint: Convert list to LinkedHashSet, then back to list.
    Expected: List contains unique elements, order preserved.
14. **Check if LinkedHashSet contains a number.**
    Hint: Use `contains()`.
    Expected: `true` or `false`.
15. **Add element at end (LinkedHashSet) — effect?**
    Hint: Use `add()`.
    Expected: Element appended, preserves order.
16. **Remove a specific element from LinkedHashSet.**
    Hint: Use `remove()`.
    Expected: Element deleted, order of others preserved.
17. **Find size of LinkedHashSet.**
    Hint: Use `size()`.
    Expected: Integer count.
18. **Iterate LinkedHashSet using Iterator.**
    Hint: Use `Iterator<Integer>` and `hasNext()`.
    Expected: Elements printed in insertion order.
19. **Union of two HashSets.**
    Hint: Use `addAll()` method.
    Expected: Combined set with unique elements.
20. **Intersection of two HashSets.**
    Hint: Use `retainAll()` method.
    Expected: Set contains only common elements.

# 4) TreeSet — 15 Questions

1. **Create a TreeSet of integers and add 5 numbers.**
   Hint: Use `TreeSet<Integer>`; elements auto-sorted.
   Expected: Set stores numbers in ascending order.
2. **Print all elements of TreeSet.**
   Hint: Use for-each loop.
   Expected: Elements printed in ascending order.
3. **Check if TreeSet contains a number (e.g., 10).**
   Hint: Use `contains()`.
   Expected: `true` or `false`.
4. **Remove an element from TreeSet.**
   Hint: Use `remove()`.
   Expected: Element deleted; order maintained.
5. **Find first (smallest) element in TreeSet.**
   Hint: Use `first()`.
   Expected: Returns smallest number.
6. **Find last (largest) element in TreeSet.**
   Hint: Use `last()`.
   Expected: Returns largest number.
7. **Remove first element.**
   Hint: Use `pollFirst()`.
   Expected: Smallest element removed.
8. **Remove last element.**
   Hint: Use `pollLast()`.
   Expected: Largest element removed.
9. **Find numbers greater than a value (e.g., 10).**
   Hint: Use `tailSet(10, false)`.
   Expected: Set of numbers > 10.
10. **Find numbers less than a value (e.g., 20).**
    Hint: Use `headSet(20, false)`.
    Expected: Set of numbers < 20.
11. **Find subset between two numbers (10–30).**
    Hint: Use `subSet(10, true, 30, true)`.
    Expected: Set of numbers from 10 to 30 inclusive.
12. **Convert TreeSet to ArrayList.**
    Hint: Use `new ArrayList<>(treeSet)`.
    Expected: List contains sorted elements.
13. **Add duplicate element to TreeSet.**
    Hint: Use `add()`.
    Expected: Duplicate ignored; order unchanged.
14. **Iterate TreeSet using Iterator.**
    Hint: Use `Iterator<Integer>`.
    Expected: Elements printed in ascending order.
15. **Find size of TreeSet.**
    Hint: Use `size()`.
    Expected: Integer representing number of unique elements.

# 5) HashMap — 40 Questions

1. **Create a HashMap with Integer keys and String values.**
   Hint: Use `HashMap<Integer, String>` and `put()`.
   Expected: Map stores key-value pairs.
2. **Add 5 key-value pairs to HashMap.**
   Hint: Use `put(key, value)`.
   Expected: Map contains all 5 pairs.
3. **Print all keys of HashMap.**
   Hint: Use `keySet()`.
   Expected: Set of keys printed.
4. **Print all values of HashMap.**
   Hint: Use `values()`.
   Expected: Collection of values printed.
5. **Print all key-value pairs.**
   Hint: Use `entrySet()`.
   Expected: Each key-value pair printed.
6. **Check if a key exists (e.g., 10).**
   Hint: Use `containsKey()`.
   Expected: `true` or `false`.
7. **Check if a value exists (e.g., "Java").**
   Hint: Use `containsValue()`.
   Expected: `true` or `false`.
8. **Remove a key-value pair using key.**
   Hint: Use `remove(key)`.
   Expected: Pair deleted if key exists.
9. **Find size of HashMap.**
   Hint: Use `size()`.
   Expected: Number of key-value pairs.
10. **Clear all entries in HashMap.**
    Hint: Use `clear()`.
    Expected: Map becomes empty.
11. **Iterate HashMap using for-each on entrySet.**
    Hint: Use `for(Map.Entry<K,V> entry: map.entrySet())`.
    Expected: Each key-value pair printed.
12. **Iterate HashMap using for-each on keySet.**
    Hint: Use `get(key)` inside loop.
    Expected: Key-value pairs printed.
13. **Update value of a key.**
    Hint: Use `put(key, newValue)`.
    Expected: Old value replaced with new.
14. **Merge two HashMaps.**
    Hint: Use `putAll()`.
    Expected: First map now contains all entries from second.

15. **Check if HashMap is empty.**
    Hint: Use `isEmpty()`.
    Expected: `true` if no entries.
16. **Count frequency of characters in a string using HashMap.**
    Hint: Iterate string, update counts using `getOrDefault()`.
    Expected: Map of character → frequency.
17. **Count frequency of numbers in array using HashMap.**
    Hint: Similar to previous, but iterate array.
    Expected: Map of number → count.
18. **Find the key with maximum value.**
    Hint: Iterate entrySet, track max.
    Expected: Key whose value is largest.
19. **Find the key with minimum value.**
    Hint: Iterate entrySet, track min.
    Expected: Key whose value is smallest.
20. **Remove all keys with value less than a threshold.**
    Hint: Iterate and remove using `remove(key)` or iterator.
    Expected: Only keys with value ≥ threshold remain.
21. **Swap keys and values in HashMap.**
    Hint: Create new map and invert entries.
    Expected: Keys become values, values become keys.
22. **Print top 3 keys with highest values.**
    Hint: Use entrySet + sorting.
    Expected: Three keys printed.
23. **Print all keys in ascending order.**
    Hint: Convert keySet to list and sort.
    Expected: Keys printed ascending.
24. **Print all values in ascending order.**
    Hint: Convert values collection to list and sort.
    Expected: Values printed ascending.
25. **Find if two HashMaps are equal.**
    Hint: Use `equals()`.
    Expected: `true` if all entries match.
26. **Increment value of a key by 1 if exists, else add with value 1.**
    Hint: Use `getOrDefault()`.
    Expected: Counts updated properly.
27. **Remove duplicate values (keep first key only).**
    Hint: Use a `HashSet` for values while iterating.
    Expected: Only first occurrence of value remains.
28. **Check if two strings are anagrams using HashMap.**
    Hint: Count character frequency for both strings.
    Expected: `true` if frequencies match.
29. **Group words by their length.**
    Hint: Map from length → List of words.
    Expected: Words grouped correctly.

30. **Find first non-repeating character using HashMap.**
    Hint: Count frequency, then check string order.
    Expected: Returns first unique character.
31. **Find first repeating character using HashMap.**
    Hint: Count frequency, then iterate string.
    Expected: Returns first repeated character.
32. **Find all keys with even values.**
    Hint: Iterate entrySet and check value.
    Expected: Print keys with even value.
33. **Check if two arrays have same frequency of elements using HashMap.**
    Hint: Count frequencies for both arrays.
    Expected: `true` if maps match.
34. **Print values in descending order.**
    Hint: Convert values to list and sort reverse.
    Expected: Values printed descending.
35. **Find maximum frequency of any value.**
    Hint: Track max while iterating entrySet.
    Expected: Integer = max frequency.
36. **Remove all keys greater than a given number.**
    Hint: Iterate keys and remove selectively.
    Expected: Only keys ≤ given number remain.
37. **Check if a list of numbers contains duplicates using HashMap.**
    Hint: Count frequencies, check any >1.
    Expected: `true` if duplicates exist.
38. **Count number of distinct elements using HashMap.**
    Hint: Keys of frequency map.
    Expected: Number of unique elements.
39. **Sort HashMap by values ascending.**
    Hint: Convert entrySet to list and sort by value.
    Expected: Entry list sorted by value.
40. **Sort HashMap by values descending.**
    Hint: Similar to above, sort reverse.
    Expected: Entry list sorted descending by value.

# 6) LinkedHashMap — 10 Questions

1. **Create a LinkedHashMap with Integer keys and String values.**
   Hint: Use `LinkedHashMap<Integer, String>`; insertion order preserved.
   Expected: Map stores key-value pairs in order added.
2. **Add 5 key-value pairs to LinkedHashMap.**
   Hint: Use `put(key, value)`.
   Expected: Keys maintain insertion order.
3. **Print all keys and values.**
   Hint: Use `entrySet()` with for-each.
   Expected: Pairs printed in insertion order.

4. **Check if a key exists (e.g., 10).**
   Hint: Use `containsKey()`.
   Expected: `true` or `false`.
5. **Check if a value exists (e.g., "Java").**
   Hint: Use `containsValue()`.
   Expected: `true` or `false`.
6. **Update a value for a key.**
   Hint: Use `put(key, newValue)`.
   Expected: Value replaced; order unchanged.
7. **Remove a key-value pair.**
   Hint: Use `remove(key)`.
   Expected: Pair deleted; order preserved.
8. **Iterate using for-each on keySet.**
   Hint: Use `get(key)` inside loop.
   Expected: Pairs printed in insertion order.
9. **Iterate using entrySet iterator.**
   Hint: Use `Iterator<Map.Entry<K,V>>`.
   Expected: Pairs printed in insertion order.
10. **Sort LinkedHashMap by values.**
    Hint: Convert entrySet to list, sort, then create new LinkedHashMap.
    Expected: Map entries sorted by value; iteration preserves this new order.

---

# 7) TreeMap — 15 Questions

1. **Create a TreeMap with Integer keys and String values.**
   Hint: Use `TreeMap<Integer, String>`; keys auto-sorted.
   Expected: Entries stored in ascending key order.
2. **Add 5 key-value pairs.**
   Hint: Use `put(key, value)`.
   Expected: Keys sorted automatically.
3. **Print all keys and values.**
   Hint: Use `entrySet()` or for-each on keySet.
   Expected: Pairs printed ascending by key.
4. **Check if a key exists (e.g., 10).**
   Hint: Use `containsKey()`.
   Expected: `true` or `false`.
5. **Check if a value exists (e.g., "Java").**
   Hint: Use `containsValue()`.
   Expected: `true` or `false`.
6. **Remove a key-value pair.**
   Hint: Use `remove(key)`.
   Expected: Pair removed; order maintained.

7. **Find first (smallest) key.**
   Hint: Use `firstKey()`.
   Expected: Returns smallest key.
8. **Find last (largest) key.**
   Hint: Use `lastKey()`.
   Expected: Returns largest key.
9. **Get all keys less than a value (e.g., 20).**
   Hint: Use `headMap(20)`.
   Expected: Map of keys < 20.
10. **Get all keys greater than or equal to a value (e.g., 10).**
    Hint: Use `tailMap(10)`.
    Expected: Map with keys ≥ 10.
11. **Get subset between two keys (e.g., 10–30).**
    Hint: Use `subMap(10, true, 30, true)`.
    Expected: Map with keys 10 to 30 inclusive.
12. **Iterate TreeMap using for-each on entrySet.**
    Hint: Use `Map.Entry<K,V>`.
    Expected: Pairs printed in key order.
13. **Find key with maximum value.**
    Hint: Iterate entrySet and track max value.
    Expected: Key corresponding to largest value.
14. **Sort TreeMap by values.**
    Hint: Convert entrySet to list and sort by value.
    Expected: Sorted list of entries by value.
15. **Convert TreeMap to ArrayList of keys.**
    Hint: Use `new ArrayList<>(treeMap.keySet())`.
    Expected: List of keys in ascending order.

# 8) Queue (ArrayDeque) — 15 Questions

1. **Create a Queue using `ArrayDeque` and add 5 integers.**
   Hint: Use `ArrayDeque<Integer>` and `add()`.
   Expected: Queue stores elements in insertion order.
2. **Print all elements of the Queue.**
   Hint: Use for-each loop.
   Expected: Elements printed in insertion order.
3. **Remove element from front of Queue.**
   Hint: Use `poll()` or `remove()`.
   Expected: First element removed; others shift forward.
4. **Peek front element without removing.**
   Hint: Use `peek()`.
   Expected: Returns first element.
5. **Check if Queue contains a number (e.g., 10).**
   Hint: Use `contains()`.
   Expected: `true` or `false`.

6. **Get size of Queue.**
   Hint: Use `size()`.
   Expected: Number of elements.
7. **Clear all elements from Queue.**
   Hint: Use `clear()`.
   Expected: Queue becomes empty.
8. **Add element at the end.**
   Hint: Use `offer()` or `add()`.
   Expected: Element appended.
9. **Iterate Queue using iterator.**
   Hint: Use `Iterator<Integer>`.
   Expected: Elements printed in order.
10. **Convert Queue to ArrayList.**
    Hint: Use `new ArrayList<>(queue)`.
    Expected: List contains same elements.
11. **Remove all even numbers from Queue.**
    Hint: Use iterator and `%2` check.
    Expected: Queue contains only odd numbers.
12. **Print first and last elements of Queue.**
    Hint: Use `peek()` for front, `peekLast()` for rear.
    Expected: First and last values returned.
13. **Add duplicate elements to Queue.**
    Hint: Use `add()` multiple times.
    Expected: Queue stores all elements including duplicates.
14. **Check if Queue is empty.**
    Hint: Use `isEmpty()`.
    Expected: `true` if empty.
15. **Reverse elements of Queue.**
    Hint: Convert to list, reverse, add back.
    Expected: Queue elements in reverse order.

---

# 9) PriorityQueue — 15 Questions

1. **Create a PriorityQueue of integers.**
   Hint: Use `PriorityQueue<Integer>`.
   Expected: Queue maintains natural ordering (min-heap).
2. **Add 5 integers to PriorityQueue.**
   Hint: Use `add()` or `offer()`.
   Expected: Elements arranged in heap order.
3. **Print elements of PriorityQueue.**
   Hint: Use iterator (order not guaranteed).
   Expected: All elements printed.

4. **Peek minimum element without removing.**
   Hint: Use `peek()`.
   Expected: Returns smallest element.
5. **Remove minimum element.**
   Hint: Use `poll()`.
   Expected: Smallest element removed.
6. **Check if PriorityQueue contains a value (e.g., 10).**
   Hint: Use `contains()`.
   Expected: `true` or `false`.
7. **Get size of PriorityQueue.**
   Hint: Use `size()`.
   Expected: Integer size.
8. **Clear PriorityQueue.**
   Hint: Use `clear()`.
   Expected: Queue becomes empty.
9. **Iterate PriorityQueue using iterator.**
   Hint: Use `Iterator<Integer>`.
   Expected: All elements printed (heap order, not sorted).
10. **Convert PriorityQueue to ArrayList.**
    Hint: Use `new ArrayList<>(pq)`.
    Expected: List contains elements (may need sorting if order matters).
11. **Add custom comparator to PriorityQueue (reverse order).**
    Hint: Use `new PriorityQueue<>(Collections.reverseOrder())`.
    Expected: Queue behaves as max-heap.
12. **Remove all elements greater than a threshold.**
    Hint: Iterate using iterator and `remove()`.
    Expected: Only elements ≤ threshold remain.
13. **Find top 3 largest elements in PriorityQueue.**
    Hint: Poll three times from max-heap.
    Expected: Three largest numbers returned.
14. **Check if PriorityQueue is empty.**
    Hint: Use `isEmpty()`.
    Expected: `true` if empty.
15. **Merge two PriorityQueues.**
    Hint: Use `addAll()`.
    Expected: Single queue contains all elements in heap order.

# 10) Iterator & ListIterator — 10 Questions

1. **Create an ArrayList and obtain an Iterator.**
   Hint: Use `list.iterator()`.
   Expected: Iterator created successfully.
2. **Traverse ArrayList using Iterator and print elements.**
   Hint: Use `hasNext()` and `next()`.
   Expected: All elements printed in order.

3. **Remove all even numbers using Iterator.**
   Hint: Use `iterator.remove()` after checking `%2==0`.
   Expected: Only odd numbers remain.
4. **Create a LinkedList and obtain ListIterator.**
   Hint: Use `list.listIterator()`.
   Expected: ListIterator created successfully.
5. **Traverse LinkedList forward using ListIterator.**
   Hint: Use `hasNext()` and `next()`.
   Expected: Elements printed in forward order.
6. **Traverse LinkedList backward using ListIterator.**
   Hint: Use `hasPrevious()` and `previous()`.
   Expected: Elements printed in reverse order.
7. **Add an element in the middle using ListIterator.**
   Hint: Use `listIterator.add(value)`.
   Expected: Element inserted at current position.
8. **Replace an element using ListIterator.**
   Hint: Use `listIterator.set(value)`.
   Expected: Current element replaced.
9. **Remove element at current position using ListIterator.**
   Hint: Use `listIterator.remove()`.
   Expected: Element deleted.
10. **Check if ListIterator has next and previous elements.**
    Hint: Use `hasNext()` and `hasPrevious()`.
    Expected: Boolean values indicating position in list.

---

# 11) Comparable & Comparator — 10 Questions

1. **Create a `Student` class with `name` and `marks` and implement `Comparable`.**
   Hint: Implement `compareTo()` based on marks.
   Expected: Students can be sorted by marks.
2. **Create an ArrayList of Students and sort using `Comparable`.**
   Hint: Use `Collections.sort(list)`.
   Expected: List sorted by marks ascending.
3. **Sort ArrayList of Students by name using Comparator.**
   Hint: Use `Collections.sort(list, Comparator.comparing(Student::getName))`.
   Expected: Students sorted alphabetically by name.
4. **Sort ArrayList of Students by marks descending using Comparator.**
   Hint: Use `Comparator.comparing(Student::getMarks).reversed()`.
   Expected: Students sorted descending by marks.

5. **Sort array of integers using custom Comparator (descending).**
   Hint: Use `Collections.sort(list, Comparator.reverseOrder())`.
   Expected: Numbers sorted descending.
6. **Sort array of strings by length using Comparator.**
   Hint: Use `Comparator.comparing(String::length)`.
   Expected: Strings sorted by increasing length.
7. **Sort array of strings by last character using Comparator.**
   Hint: Use lambda `(s1,s2) -> s1.charAt(s1.length()-1) - s2.charAt(s2.length()-1)`.
   Expected: Strings sorted by last character.
8. **Sort objects by multiple fields (marks, then name).**
   Hint: Use `Comparator.comparing(Student::getMarks).thenComparing(Student::getName)`.
   Expected: Sorted first by marks, tie-breaker by name.
9. **Sort using anonymous Comparator class.**
   Hint: Implement `compare()` inside anonymous class.
   Expected: Works same as lambda/comparator.
10. **Sort using stream and Comparator.**
    Hint: `list.stream().sorted(Comparator).collect(Collectors.toList())`.
    Expected: Returns sorted list.

# 12) Collections Utility Class — 15 Questions

1. **Sort an ArrayList in ascending order.**
   Hint: Use `Collections.sort(list)`.
   Expected: List elements sorted ascending.
2. **Sort an ArrayList in descending order.**
   Hint: Use `Collections.sort(list, Collections.reverseOrder())`.
   Expected: List elements sorted descending.
3. **Reverse an ArrayList.**
   Hint: Use `Collections.reverse(list)`.
   Expected: Elements appear in reverse order.
4. **Find maximum element in ArrayList.**
   Hint: Use `Collections.max(list)`.
   Expected: Returns largest element.
5. **Find minimum element in ArrayList.**
   Hint: Use `Collections.min(list)`.
   Expected: Returns smallest element.
6. **Swap two elements in ArrayList.**
   Hint: Use `Collections.swap(list, i, j)`.
   Expected: Elements at index `i` and `j` exchanged.
7. **Find frequency of an element in ArrayList.**
   Hint: Use `Collections.frequency(list, value)`.
   Expected: Returns count of occurrences.

8. **Copy one ArrayList to another.**
   Hint: Use `Collections.copy(dest, src)`.
   Expected: Destination list updated with source elements.
9. **Fill ArrayList with a single value.**
   Hint: Use `Collections.fill(list, value)`.
   Expected: All elements replaced by value.
10. **Replace all occurrences of an element.**
    Hint: Use `Collections.replaceAll(list, oldValue, newValue)`.
    Expected: All oldValue replaced by newValue.
11. **Rotate an ArrayList by k positions.**
    Hint: Use `Collections.rotate(list, k)`.
    Expected: List rotated to right by k steps.
12. **Shuffle elements of ArrayList randomly.**
    Hint: Use `Collections.shuffle(list)`.
    Expected: List elements in random order.
13. **Binary search in a sorted ArrayList.**
    Hint: Use `Collections.binarySearch(list, key)`.
    Expected: Returns index of key or negative if not found.
14. **Check disjoint of two collections.**
    Hint: Use `Collections.disjoint(list1, list2)`.
    Expected: `true` if no common elements.
15. **Create an unmodifiable (read-only) list.**
    Hint: Use `Collections.unmodifiableList(list)`.
    Expected: Any modification attempt throws `UnsupportedOperationException`.

# 1) Classes & Objects — 15 Questions

1. **Create a `Student` class with `name` and `age` fields and print them.**
   Hint: Use a constructor or set fields after object creation.
   Expected: Prints student's name and age.
2. **Create multiple `Student` objects and print their details.**
   Hint: Instantiate multiple objects using `new`.
   Expected: Each student's details printed separately.
3. **Add a method `greet()` in `Student` class to print a greeting.**
   Hint: Method prints `"Hello, I am " + name`.
   Expected: Greeting printed for each student.
4. **Create a parameterized constructor for `Student` to initialize fields.**
   Hint: Constructor with parameters `(String name, int age)`.
   Expected: Object fields initialized at creation.
5. **Create a default constructor for `Student` that sets default values.**
   Hint: Constructor without parameters.
   Expected: Prints default values if no arguments passed.
6. **Overload constructor to allow only name or only age.**
   Hint: Use multiple constructors with different parameter lists.
   Expected: Object can be created with name only, age only, or both.

7. **Add a method `increaseAge()` that increments age by 1.**
   Hint: Use `this.age++`.
   Expected: Age of the object increases.
8. **Use `this` keyword to differentiate local variable and field in constructor.**
   Hint: `this.name = name`.
   Expected: Fields properly initialized.
9. **Create a `Car` class with `brand` and `speed` and a method `displayInfo()`.**
   Hint: Method prints brand and speed.
   Expected: Car details printed.
10. **Create multiple objects of `Car` and print their info using a loop.**
    Hint: Store objects in an array or ArrayList.
    Expected: Details printed for each car.
11. **Create a method `accelerate(int increment)` that increases speed.**
    Hint: Add increment to `speed`.
    Expected: Speed updated correctly.
12. **Overload method `accelerate()` without parameters to increase speed by 10.**
    Hint: Use method overloading.
    Expected: Default increment works if no parameter passed.
13. **Create a class `Rectangle` with `length` and `width` and method `area()`.**
    Hint: Return `length*width`.
    Expected: Correct area returned.
14. **Overload `area()` method to accept `length` and `width` as parameters.**
    Hint: Method overloading.
    Expected: Can calculate area for different dimensions.
15. **Create a `Counter` class with a static counter variable to count objects created.**
    Hint: Increment static variable in constructor.
    Expected: Prints number of objects created.

# 2) Encapsulation — 10 Questions

1. **Create a `Person` class with private fields `name` and `age`.**
   Hint: Use `private String name; private int age;`.
   Expected: Fields not accessible directly from outside.
2. **Add public getters and setters for `name` and `age`.**
   Hint: Methods like `getName()`, `setName(String name)`.
   Expected: Access and modify private fields safely.
3. **Add validation in `setAge(int age)` to allow only 0–120.**
   Hint: Use `if(age>=0 && age<=120) this.age=age;`.
   Expected: Invalid ages are ignored.
4. **Create a `BankAccount` class with private `balance` and `accountNumber`.**
   Hint: Use getters for both, setter only for balance.
   Expected: Encapsulated fields for safe access.
5. **Add `deposit(double amount)` and `withdraw(double amount)` methods.**
   Hint: Update balance only if deposit >0 or withdraw <= balance.
   Expected: Balance updated safely.

6. **Create a read-only field `accountNumber` (no setter).**
   Hint: Only provide getter.
   Expected: Value can't be modified after initialization.
7. **Create a write-only field `password` in `BankAccount`.**
   Hint: Only setter, no getter.
   Expected: Can set password but not read it from outside.
8. **Encapsulate `email` in `Person` class with simple validation (@ present).**
   Hint: In setter, check `if(email.contains("@")) this.email=email;`.
   Expected: Invalid email rejected.
9. **Create an array of `Person` objects and update ages using setter.**
   Hint: Loop through array and call `setAge()`.
   Expected: Ages updated only if valid.
10. **Print all details of encapsulated objects using getters.**
    Hint: Access all private fields via getter methods.
    Expected: All information printed safely without exposing fields directly.

# 3) Inheritance — 20 Questions

1. **Create a class `Animal` with method `eat()`. Create a subclass `Dog` that inherits `Animal` and calls `eat()`.**
   Hint: Use `extends`.
   Expected: Dog object can call `eat()`.
2. **Override `eat()` in `Dog` to print `"Dog is eating"`.**
   Hint: Use `@Override`.
   Expected: Dog calls its own `eat()`.
3. **Add a method `bark()` in `Dog`.**
   Hint: Only Dog can call this.
   Expected: Dog barks.
4. **Create another subclass `Cat` with its own `meow()` method.**
   Hint: Inherit from Animal.
   Expected: Cat can call `eat()` and `meow()`.
5. **Create objects of `Dog` and `Cat` and demonstrate polymorphism with `Animal` reference.**
   Hint: `Animal a = new Dog(); a.eat();`
   Expected: Dog's overridden `eat()` called.
6. **Create a class `Vehicle` with fields `brand` and `speed` and method `display()`.**
   Hint: Use constructor to initialize.
   Expected: Vehicle details printed.
7. **Create subclass `Car` with additional field `fuelType` and method `displayCar()`.**
   Hint: Use `super` to call parent constructor.
   Expected: Vehicle + Car details printed.
8. **Create subclass `Bike` with method `displayBike()` demonstrating `super`.**
   Hint: Call parent method using `super.display()`.
   Expected: Bike details printed with vehicle info.

9. **Demonstrate multilevel inheritance: `Grandparent -> Parent -> Child` with methods in each.**
   Hint: `Child` object should call all methods.
   Expected: All inherited methods accessible.
10. **Create constructor in `Parent` and `Child`; demonstrate constructor chaining using `super()`.**
    Hint: Child constructor calls `super()`.
    Expected: Parent constructor executed before child.
11. **Create a method `calculate()` in `Parent` and override in `Child` with different logic.**
    Hint: Show method overriding.
    Expected: Child's `calculate()` called for Child object.
12. **Use `final` method in `Parent` and attempt to override in `Child`.**
    Hint: Compiler should prevent overriding.
    Expected: Error if override attempted.
13. **Demonstrate `super` keyword to access parent field.**
    Hint: Parent and Child have same field name.
    Expected: Child can access parent field via `super.field`.
14. **Create array of `Parent` references storing `Child` objects and call overridden methods.**
    Hint: Demonstrates runtime polymorphism.
    Expected: Correct overridden methods executed.
15. **Create multiple subclasses (`Car`, `Bike`, `Truck`) of `Vehicle` and demonstrate method overriding.**
    Hint: Each overrides `display()`.
    Expected: Calls correct `display()` based on object type.
16. **Create `Shape` class with `area()` method and subclasses `Circle`, `Rectangle`, `Triangle` overriding `area()`.**
    Hint: Use proper formulas in overridden methods.
    Expected: Correct area computed for each shape.
17. **Demonstrate typecasting with parent reference to child object.**
    Hint: `(Child) parentRef`.
    Expected: Access child-specific methods.
18. **Create a class `Employee` and subclass `Manager` adding bonus calculation.**
    Hint: Call parent constructor with `super()`.
    Expected: Salary + bonus printed.
19. **Demonstrate hierarchical inheritance: `Employee -> Manager`, `Employee -> Developer` with separate methods.**
    Hint: Parent class shared by multiple children.
    Expected: Each child calls own methods + inherited methods.
20. **Create method in `Parent` that calls another overridden method in `Child` using `super` in combination.**
    Hint: Demonstrates `super` and overriding together.
    Expected: Parent logic + child-specific logic executed correctly.

# 4) Polymorphism — 15 Questions

**Compile-Time Polymorphism (Method Overloading) — 8 Questions**

1. **Create a class `Calculator` with method `add(int a, int b)` and overload it with `add(int a, int b, int c)`.**
   Hint: Method name same, different parameters.
   Expected: Correct sum returned based on arguments.
2. **Overload `add(double a, double b)` in `Calculator`.**
   Hint: Demonstrates overloading with different types.
   Expected: Correct sum for doubles.
3. **Overload a method `greet()` with no parameters and with `greet(String name)`.**
   Hint: Two methods same name, different parameters.
   Expected: Prints generic or personalized greeting.
4. **Overload method `multiply(int a, int b)` and `multiply(int[] arr)`.**
   Hint: Demonstrates overloading with array input.
   Expected: Returns correct product.
5. **Overload `display()` method with String, int, and boolean parameters.**
   Hint: Each version prints different types.
   Expected: Correct method invoked automatically.
6. **Overload a method with varargs `sum(int... numbers)`.**
   Hint: Accepts any number of integers.
   Expected: Returns total sum.
7. **Overload `calculate(int a, double b)` and `calculate(double a, int b)`.**
   Hint: Different parameter order.
   Expected: Correct method invoked based on argument types.
8. **Overload constructor in a class `Box` for different dimensions.**
   Hint: Constructor overloading.
   Expected: Different Box objects created based on parameters.

---

**Runtime Polymorphism (Method Overriding) — 7 Questions**

9. **Create class `Animal` with method `sound()`, override in `Dog` and `Cat`.**
   Hint: Use `@Override`.
   Expected: Correct sound printed for each object.
10. **Demonstrate runtime polymorphism using parent reference: `Animal a = new Dog(); a.sound();`**
    Hint: Parent reference can hold child object.
    Expected: Dog's sound printed.
11. **Override a method and call parent method using `super.method()` inside child.**
    Hint: Combine parent + child logic.
    Expected: Both parent and child outputs displayed.
12. **Create `Vehicle` class with `start()` method, override in `Car` and `Bike`.**
    Hint: Demonstrate different behaviors.
    Expected: Correct start message for each type.

13. **Use an array of parent type `Animal[]` storing multiple child objects and call overridden method.**
    Hint: Demonstrates dynamic method dispatch.
    Expected: Correct method executed for each object.
14. **Override a method in subclass and call it multiple times with parent reference.**
    Hint: Test runtime behavior.
    Expected: Always child's overridden method executed.
15. **Demonstrate overriding with return type covariance (return subclass type).**
    Hint: Parent method returns `Animal`, child returns `Dog`.
    Expected: Works correctly in Java 1.5+.

# 5) Abstraction — 15 Questions

**Abstract Classes — 8 Questions**

1. **Create an abstract class `Shape` with abstract method `area()`.**
   Hint: `abstract class Shape { abstract double area(); }`
   Expected: Cannot instantiate Shape directly.
2. **Create subclass `Circle` with field `radius` and implement `area()`.**
   Hint: `return Math.PI * radius * radius;`
   Expected: Circle area calculated correctly.
3. **Create subclass `Rectangle` with `length` and `width` and implement `area()`.**
   Hint: Override `area()`.
   Expected: Returns correct rectangle area.
4. **Create an array of `Shape` references storing different shapes and call `area()`.**
   Hint: Demonstrates runtime polymorphism with abstract class.
   Expected: Correct area computed for each shape.
5. **Add a concrete method `display()` in abstract class `Shape` that prints "I am a shape".**
   Hint: Mix abstract + concrete methods.
   Expected: Child can call concrete method without redefining.
6. **Demonstrate constructor in abstract class and call it from subclass.**
   Hint: Abstract class can have constructor, called via `super()`.
   Expected: Constructor executed correctly.
7. **Create abstract method `perimeter()` and implement in child classes.**
   Hint: Each shape provides its own logic.
   Expected: Correct perimeter returned.
8. **Create a subclass `Square` extending `Rectangle` and override methods if needed.**
   Hint: Demonstrates multilevel inheritance with abstract class.
   Expected: Correct area and perimeter for Square.

---

**Interfaces — 7 Questions**

9. **Create interface `Movable` with method `move()`.**
   Hint: `interface Movable { void move(); }`
   Expected: Cannot instantiate interface.
10. **Implement `Movable` in class `Car` and override `move()` to print `"Car is moving"`.**
    Hint: Use `implements` keyword.
    Expected: Car object can move.
11. **Implement multiple interfaces `Movable` and `Stopable` in a class `Bike`.**
    Hint: Use `implements Movable, Stopable`.
    Expected: Bike must override all interface methods.
12. **Create an interface with default method `status()` and call it from implementing class.**
    Hint: Use `default` keyword in interface.
    Expected: Class can call default method without overriding.
13. **Create a static method in interface and call it directly using interface name.**
    Hint: Use `static` keyword.
    Expected: InterfaceName.method() works.
14. **Demonstrate interface reference holding implementing class object.**
    Hint: `Movable m = new Car(); m.move();`
    Expected: Correct method executed.
15. **Create multiple classes implementing same interface and call method using array of interface references.**
    Hint: Polymorphism with interfaces.
    Expected: Correct behavior executed for each object.

# 6) Static Members — 10 Questions

1. **Create a class `Counter` with a static variable `count` to count objects created.**
   Hint: Increment `count` in constructor.
   Expected: Prints number of objects created so far.
2. **Create static method `displayCount()` to print the count of objects.**
   Hint: `static void displayCount() { System.out.println(count); }`
   Expected: Can call without object.
3. **Access static variable without creating an object.**
   Hint: `Counter.count`
   Expected: Correct value printed.
4. **Demonstrate static block to initialize static variables.**
   Hint: `static { ... }` executed before main method.
   Expected: Static variable initialized before any object creation.
5. **Create a class `MathUtils` with static method `square(int n)` to return $n^2$.**
   Hint: Can call `MathUtils.square(5)` directly.
   Expected: Returns correct square.
6. **Create a class `Employee` with static field `companyName` shared by all employees.**
   Hint: Static fields shared across all objects.
   Expected: Changing in one object affects all.

7. **Demonstrate static method accessing only static fields.**
   Hint: Static methods cannot access instance fields directly.
   Expected: Compiler prevents access to non-static fields.
8. **Create a static constant using `final static` in a class.**
   Hint: `final static double PI = 3.14;`
   Expected: Value cannot be changed.
9. **Create static inner class `Outer.StaticInner` and access its method.**
   Hint: Can create instance using `new Outer.StaticInner()`.
   Expected: Method works without outer object.
10. **Count total number of method calls using a static variable.**
    Hint: Increment static variable in method body.
    Expected: Shows cumulative calls across all objects.

# 7) Final Keyword — 10 Questions

1. **Create a `final` variable and attempt to modify it.**
   Hint: `final int MAX = 100;`
   Expected: Compiler error if you try to change its value.
2. **Create a `final` method in a class and try to override it in a subclass.**
   Hint: `final void display() { ... }`
   Expected: Compiler prevents overriding.
3. **Create a `final` class and try to extend it.**
   Hint: `final class Utility { ... }`
   Expected: Compiler prevents inheritance.
4. **Use `final` variable with object reference and modify object fields.**
   Hint: `final Employee e = new Employee(); e.name = "John";`
   Expected: Object fields can change, but reference cannot point to new object.
5. **Use `final` in method parameter and try to modify it.**
   Hint: `void greet(final String name) { name = "abc"; }`
   Expected: Compiler error.
6. **Demonstrate `final` variable in a loop.**
   Hint: `for(final int i=0; i<5; i++)`
   Expected: Compiler prevents incrementing final variable.
7. **Create `final static` constant and access it from another class.**
   Hint: `ClassName.CONSTANT`
   Expected: Value accessible but cannot be modified.
8. **Create immutable class using `final` class and `final` fields.**
   Hint: All fields private and no setters.
   Expected: Object state cannot be changed after creation.
9. **Use `final` with a local variable inside a method.**
   Hint: Variable cannot be reassigned after initialization.
   Expected: Compiler prevents reassignment.
10. **Use `final` with method return type reference.**
    Hint: Demonstrates final reference type.
    Expected: Reference cannot change but object content can if mutable.

# 8) Object Class Methods — 10 Questions

1. **Override `toString()` in `Student` class to print name and age.**
   Hint: `public String toString() { return name + " " + age; }`
   Expected: Printing object shows readable details.
2. **Create two `Student` objects and compare using `equals()` without overriding.**
   Hint: Default `equals()` compares references.
   Expected: Returns false for two different objects.
3. **Override `equals()` in `Student` to compare name and age.**
   Hint: Use `instanceof` and field comparison.
   Expected: Objects with same data are considered equal.
4. **Override `hashCode()` consistent with `equals()`.**
   Hint: `return Objects.hash(name, age);`
   Expected: Equal objects have same hash code.
5. **Demonstrate difference between == and `equals()` for objects.**
   Hint: `==` compares references, `equals()` compares content.
   Expected: Shows difference clearly.
6. **Create a `clone()` method in `Student` using `Cloneable` interface.**
   Hint: Implement `Cloneable` and override `clone()`.
   Expected: Object cloned successfully.
7. **Demonstrate shallow copy using `clone()`.**
   Hint: Modifying nested object affects original.
   Expected: Shows shallow copy behavior.
8. **Demonstrate deep copy manually in `clone()` method.**
   Hint: Create new nested objects in clone.
   Expected: Modifications in clone do not affect original.
9. **Print class name of an object using `getClass()`.**
   Hint: `obj.getClass().getName()`.
   Expected: Returns full class name.
10. **Demonstrate `finalize()` method (deprecated but for learning).**
    Hint: Override `protected void finalize()`.
    Expected: Called before object is garbage collected.

# 9) Inner Classes — 10 Questions

1. **Create a member inner class `Engine` inside class `Car` and access its method.**
   Hint: `Car.Engine e = car.new Engine();`
   Expected: Engine method executed using Car object.
2. **Create a static inner class `Engine` and access its method without outer object.**
   Hint: `Car.Engine e = new Car.Engine();`
   Expected: Works without creating Car object.
3. **Create a local inner class inside a method and access it within the method.**
   Hint: Class defined inside a method body.
   Expected: Can only use inside that method.

4. **Create an anonymous inner class implementing interface `Movable`.**
   Hint: `Movable m = new Movable() { public void move() {...} };`
   Expected: Anonymous class created and method works.
5. **Use an anonymous inner class to extend an abstract class `Shape` and implement `area()`.**
   Hint: `Shape s = new Shape(){ public double area(){...} };`
   Expected: Object created and area computed.
6. **Access outer class variable from inner class.**
   Hint: `Outer.this.field`
   Expected: Inner class can access outer field.
7. **Access outer class method from inner class.**
   Hint: Call `Outer.this.method()`.
   Expected: Method executed correctly.
8. **Create inner class with same field name as outer class and use `this` to differentiate.**
   Hint: `this.field` vs `Outer.this.field`
   Expected: Correct fields accessed.
9. **Create static inner class with static method.**
   Hint: `Car.Engine.printType();`
   Expected: Method can be called without object.
10. **Use inner class object inside outer class method and return it.**
    Hint: Outer method returns `new Inner()`.
    Expected: Outer class can provide inner class objects.

# 10) Aggregation & Composition — 15 Questions

1. **Create a class `Address` and a class `Person` having `Address` as a field (Aggregation).**
   Hint: `Person` can exist without `Address`.
   Expected: Person object created with or without Address.
2. **Create `Person` class that instantiates `Address` internally (Composition).**
   Hint: `Address addr = new Address(...);` inside Person constructor.
   Expected: Address created only as part of Person object.
3. **Create multiple `Person` objects sharing the same `Address` (Aggregation).**
   Hint: Same Address object passed to multiple Persons.
   Expected: Changes in Address reflected for all persons.
4. **Demonstrate composition with `Car` having `Engine`.**
   Hint: Engine created inside Car constructor.
   Expected: Engine cannot exist without Car.
5. **Create `Company` class with a list of `Employee` objects (Aggregation).**
   Hint: Employees can exist outside Company.
   Expected: Company manages multiple employees.
6. **Create `Library` class with `Book` objects (Composition).**
   Hint: Books created inside Library.
   Expected: Books tied to library; deleted when library deleted.

7. **Access inner object's methods via outer object (Person -> Address).**
   Hint: `person.getAddress().getCity().`
   Expected: Inner object method called correctly.
8. **Modify inner object in aggregation and show effect on multiple outer objects.**
   Hint: Shared Address modified.
   Expected: Change visible in all referencing objects.
9. **Demonstrate deep composition: `University -> Department -> Professor.`**
   Hint: Each object created inside the parent.
   Expected: Objects exist only as part of hierarchy.
10. **Demonstrate shallow composition: outer object holds reference passed from outside.**
    Hint: Parent has reference, child created externally.
    Expected: Shows difference from deep composition.
11. **Create getter methods for inner objects and access their fields.**
    Hint: Encapsulation in aggregation/composition.
    Expected: Fields accessed safely.
12. **Pass inner object via constructor for aggregation.**
    Hint: Outer class stores reference.
    Expected: Works for externally created objects.
13. **Create method in outer class that modifies inner object state.**
    Hint: `person.getAddress().setCity("New City");`
    Expected: Changes reflected in inner object.
14. **Demonstrate nested aggregation: `Team -> Player -> Equipment.`**
    Hint: Multiple levels of aggregation.
    Expected: Objects linked appropriately.
15. **Demonstrate nested composition: `Car -> Engine -> Piston.`**
    Hint: All objects created internally.
    Expected: Piston cannot exist without Engine and Car.

# 11) Real-World Practice Problems — 40 Questions

**Bank Management System — 10 Questions**

1. **Create `BankAccount` class with `accountNumber`, `balance`, and `accountHolderName`.**
   Hint: Use constructor to initialize.
   Expected: Prints account details.
2. **Add methods `deposit(double amount)` and `withdraw(double amount).`**
   Hint: Update balance accordingly.
   Expected: Balance updated correctly with each operation.
3. **Add validation in withdraw to prevent overdraft.**
   Hint: Only allow if `amount <= balance`.
   Expected: Cannot withdraw more than balance.

4. **Create multiple `BankAccount` objects and store in an array.**
   Hint: Array or ArrayList.
   Expected: Can perform operations on multiple accounts.
5. **Implement `transfer(BankAccount from, BankAccount to, double amount).`**
   Hint: Withdraw from one, deposit in another.
   Expected: Amount transferred successfully.
6. **Demonstrate encapsulation by making balance private and providing getter only.**
   Hint: Users cannot modify balance directly.
   Expected: Safe access to balance.
7. **Create abstract class `Account` and inherit `SavingsAccount` and `CurrentAccount`.**
   Hint: Abstract method `calculateInterest()`.
   Expected: Each account type calculates interest differently.
8. **Add static variable to count total accounts created.**
   Hint: Increment in constructor.
   Expected: Total account count accessible.
9. **Implement method overriding: `displayDetails()` in each subclass.**
   Hint: Shows type-specific info.
   Expected: Correct details displayed per account type.
10. **Use interface `Transaction` with method `printTransactionHistory()` and implement in accounts.**
    Hint: Each account prints its own transaction list.
    Expected: Interface ensures consistent transaction logging.

---

**Library Management System — 10 Questions**

11. **Create `Book` class with `title`, `author`, `isbn`.**
    Hint: Constructor to initialize fields.
    Expected: Book object created.
12. **Create `Library` class that holds an ArrayList of `Book` objects (Composition).**
    Hint: Books created inside Library.
    Expected: Library manages its books.
13. **Add method `addBook(Book b)` and `removeBook(String isbn).`**
    Hint: Add or remove book from list.
    Expected: Correctly modifies library collection.
14. **Add method `searchByTitle(String title)` to find books.**
    Hint: Loop through ArrayList.
    Expected: Returns matching book(s).
15. **Add method `displayAllBooks().`**
    Hint: Print details of all books.
    Expected: All books listed.
16. **Create `Member` class with fields `memberId`, `name` and aggregate with Library.**
    Hint: Member borrows books.
    Expected: Library knows which members exist.

17. **Add method `borrowBook(Member m, String isbn)` in Library.**
    Hint: Track borrowed books.
    Expected: Updates library and member records.
18. **Add method `returnBook(Member m, String isbn).`**
    Hint: Reverses borrow process.
    Expected: Book available again.
19. **Demonstrate inheritance: `DigitalBook` and `PrintedBook` extending `Book`.**
    Hint: Each type has additional fields.
    Expected: Overridden methods handle specifics.
20. **Demonstrate polymorphism: `Book` reference holding `DigitalBook` or `PrintedBook`.**
    Hint: `Book b = new DigitalBook();`
    Expected: Correct overridden methods executed.

---

**Employee Management System — 10 Questions**

21. **Create `Employee` class with `id`, `name`, `salary`.**
    Hint: Constructor + getters/setters.
    Expected: Employee object created.
22. **Create subclass `Manager` with additional field `bonus`.**
    Hint: Inherit Employee.
    Expected: Manager has salary + bonus.
23. **Override method `calculateSalary()` in Manager to include bonus.**
    Hint: Shows method overriding.
    Expected: Correct salary calculated.
24. **Create subclass `Developer` with `overtimeHours` and calculate total salary.**
    Hint: Override `calculateSalary()`.
    Expected: Developer salary computed correctly.
25. **Store multiple Employee objects in ArrayList and display their salaries.**
    Hint: Use loop to call `calculateSalary()`.
    Expected: Salaries displayed for all employees.
26. **Demonstrate static variable `totalEmployees` to count employees.**
    Hint: Increment in constructor.
    Expected: Shows total number of employees.
27. **Use encapsulation to make salary private with getter and setter.**
    Hint: Validations in setter if needed.
    Expected: Safe access/modification of salary.
28. **Demonstrate aggregation: `Department` class holding multiple Employees.**
    Hint: Employees exist outside Department.
    Expected: Department manages employee list.
29. **Add method `displayDepartmentDetails()` showing all employee details.**
    Hint: Loop through employees.
    Expected: Prints department info.
30. **Demonstrate polymorphism: Employee reference pointing to Manager/Developer objects.**

Hint: `Employee e = new Manager();`
Expected: Correct overridden methods executed.

---

**Vehicle Management System — 10 Questions**

31. **Create class `Vehicle` with fields `brand`, `speed`, `color`.**
    Hint: Constructor + display method.
    Expected: Vehicle created with details.
32. **Create subclasses `Car`, `Bike`, `Truck` overriding `display()` method.**
    Hint: Each prints specific type info.
    Expected: Correct info displayed.
33. **Add method `start()` in Vehicle and override in subclasses.**
    Hint: Demonstrate method overriding.
    Expected: Correct start message.
34. **Use abstract class `Vehicle` and implement `calculateFuelConsumption()` in subclasses.**
    Hint: Each type has different logic.
    Expected: Fuel consumption calculated correctly.
35. **Demonstrate composition: `Car` has `Engine` object created internally.**
    Hint: Engine cannot exist without Car.
    Expected: Engine accessible through Car only.
36. **Demonstrate aggregation: `Fleet` class holds multiple Vehicles.**
    Hint: Vehicles can exist outside Fleet.
    Expected: Fleet manages vehicle list.
37. **Use static variable `totalVehicles` to count all vehicle objects.**
    Hint: Increment in constructor.
    Expected: Displays total vehicles.
38. **Use interface `Movable` with method `move()` implemented by all vehicle types.**
    Hint: Interface enforces common behavior.
    Expected: Polymorphic calls work.
39. **Store multiple vehicle objects in an array and call `start()` and `move()` using polymorphism.**
    Hint: Array of Vehicle references.
    Expected: Correct subclass methods executed.
40. **Demonstrate encapsulation: make `speed` private and provide getter/setter with validation.**
    Hint: Speed cannot exceed a limit.
    Expected: Safe access and modification.

# 1) Exception Handling — 12 Questions

1. **Write a program to divide two numbers and handle `ArithmeticException`.**
   Hint: Use try-catch.

2. **Handle multiple exceptions: divide by zero and array out-of-bounds.**
   Hint: `catch (ArithmeticException | ArrayIndexOutOfBoundsException e)`.
3. **Create a method that throws `NumberFormatException` when converting a string to int.**
   Hint: Use `Integer.parseInt()`.
4. **Write a program using `finally` block to close resources.**
   Hint: Even if exception occurs, `finally` executes.
5. **Use `throw` to throw a custom exception `InvalidAgeException` if age < 18.**
6. **Use `throws` to propagate exception from one method to another.**
   Hint: Method signature includes `throws Exception`.
7. **Demonstrate nested try-catch blocks.**
8. **Create a program with `try-with-resources` to read a file safely.**
9. **Write a program that catches `InputMismatchException` while reading integer input.**
10. **Create custom checked exception `InsufficientBalanceException` for BankAccount.**
11. **Demonstrate re-throwing an exception after catching.**
12. **Create a program where multiple exceptions are caught in descending order of hierarchy.**

---

# 3) Generics — 10 Questions

1. **Create a generic class `Box<T>` that stores an object of type T.**
2. **Create a generic method `printArray(T[] arr)` that prints any array.**
3. **Create a generic class `Pair<K, V>` to store key-value pairs.**
4. **Use bounded type parameter `<T extends Number>` to calculate sum of numbers.**
5. **Write a generic method to find the maximum of three values of any comparable type.**
6. **Create a generic interface `Repository<T>` with methods `add(T obj)`, `remove(T obj)`.**
7. **Create a generic class with wildcard: `List<? extends Number>` and iterate over it.**
8. **Create a generic method to swap elements of an array.**
9. **Create a generic Stack class and implement push/pop methods.**
10. **Combine Generics with Collections: `ArrayList<T>` with generic type and custom methods.**

---

# 4) Lambda Expressions & Functional Interfaces — 10 Questions

1. **Create a `Runnable` using lambda to print numbers 1–10 in a thread.**

2. **Use `Predicate<Integer>` to filter numbers > 10 from a list.**
3. **Use `Consumer<String>` to print each string in uppercase.**
4. **Use `Function<String, Integer>` to convert string to its length.**
5. **Use `Supplier<Double>` to generate a random number.**
6. **Sort a list of integers using `Comparator` and lambda expression.**
7. **Filter a list of strings to include only those starting with 'A' using lambda.**
8. **Write a method that accepts `Function<Integer, Integer>` and applies it to a list of numbers.**
9. **Use `BiFunction<Integer, Integer, Integer>` to calculate sum of two numbers.**
10. **Combine multiple lambdas: filter, map, and print using functional interfaces.**

---

# 5) Streams API — 12 Questions

1. **Create a list of integers and print even numbers using streams.**
2. **Filter strings with length > 3 from a list using streams.**
3. **Map a list of integers to their squares and collect into a new list.**
4. **Use `reduce()` to sum all numbers in a list.**
5. **Sort a list of strings in ascending order using streams.**
6. **Count the number of elements greater than 10 using streams.**
7. **Use `distinct()` to remove duplicates from a list.**
8. **Find the maximum and minimum number in a list using streams.**
9. **Group a list of strings by their length using `Collectors.groupingBy()`.**
10. **Check if all numbers in a list are positive using `allMatch()`.**
11. **Check if any string in a list contains "Java" using `anyMatch()`.**
12. **Convert list of strings to uppercase using streams and collect to list.**

---

# 6) Multithreading & Concurrency — 10 Questions

1. **Create a class that extends `Thread` and prints numbers 1–10.**
2. **Create a class that implements `Runnable` and prints numbers 1–10.**
3. **Create and start multiple threads using both approaches.**
4. **Demonstrate `join()` method to wait for threads to finish.**
5. **Demonstrate `sleep()` in a thread for 1 second between prints.**
6. **Create a synchronized method to increment a shared counter.**
7. **Create two threads accessing shared counter without synchronization and show race condition.**
8. **Use `ExecutorService` to run multiple tasks concurrently.**
9. **Use `Callable` and `Future` to return values from threads.**

10. **Demonstrate deadlock using two threads trying to lock two resources in opposite order (for learning purpose).**

# Java Patterns — 25 Questions (increasing difficulty)

### 1. Simple Star and Number Patterns — 10 Questions

1. Print a single row of 5 stars: `*****`
2. Print a column of 5 stars:

```
*
*
*
*
*
```

8. Print a right-angled triangle of stars with height 5:

```
*
**
***
****
*****
```

14. Print a right-angled triangle of numbers (row number repeated):

```
1
22
333
4444
55555
```

20. Print a square of stars (5x5):

```
*****
*****
*****
*****
*****
```

26. Print a reverse triangle of stars:

```
*****
****
***
**
*
```

32. Print numbers in a right-angled triangle incrementally:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

38. Print a pyramid of stars:

```
    *
   ***
  *****
 *******
*********
```

44. Print a reverse pyramid of stars:

```
*********
 *******
  *****
   ***
    *
```

50. Print Floyd's Triangle (numbers continuously):

```
1
2 3
4 5 6
7 8 9 10
```

---

## 2. Advanced Patterns — 15 Questions

11. Print a number pyramid (centered):

```
    1
   121
  12321
 1234321
```

16. Print a pyramid of alternating 1 and 0:

```
    1
   0 1
  1 0 1
 0 1 0 1
```

21. Print diamond of stars:

```
    *
   ***
  *****
   ***
    *
```

27. Print hollow square of stars:

```
*****
*   *
*   *
*   *
*****
```

33. Print hollow triangle of stars:

```
    *
   * *
  *   *
 *******
```

38. Print Pascal's triangle up to n=5:

```
    1
   1 1
  1 2 1
 1 3 3 1
```

1 4 6 4 1

17. Print pyramid of numbers decreasing from row number to 1:

1
2 1
3 2 1

4 3 2 1
18. Print pattern of stars in zig-zag form (height 3, width 9):
* *
* * * *
* * *
19. Print alphabet pyramid:
A
A B
A B C
A B C D
20. Print reverse alphabet triangle:
E D C B A
D C B A
C B A
B A
A
21. Print hourglass pattern of stars:
*****
***
*
***
*****
22. Print number diamond:
1
121
12321
121
1
23. Print right triangle with numbers repeated in rows:
1
2 2
3 3 3
4 4 4 4
24. Print butterfly pattern of stars (n=4):
* *
** **
*** ***
********
*** ***
** **
* *
25. Print concentric square number pattern (n=4):
4 4 4 4 4 4 4
4 3 3 3 3 3 4
4 3 2 2 2 3 4
4 3 2 1 2 3 4
4 3 2 2 2 3 4

4 3 3 3 3 3 4
4 4 4 4 4 4 4