**1. What is a data structure in Java?**

A **data structure** is a way of **organizing and storing data** so that operations (insert, search, delete, update) can be performed efficiently.

👉 In Java:

* Built-in data structures → Arrays, Strings.
* Collection Framework data structures → ArrayList, LinkedList, HashMap, HashSet, etc.
* Custom data structures → Stack, Queue, Deque, implemented using arrays or linked lists.

**Interview note**: Java’s *Collections Framework* provides ready-made implementations of most data structures.

**2. Difference between Java primitive data types and objects in collections.**

| **Feature** | **Primitives** | **Objects in Collections** |
| --- | --- | --- |
| Storage | Stored directly on the **stack** (local variables) or inside objects. | Stored as **references** on stack, actual object in **heap**. |
| Examples | int, char, double, boolean. | Integer, Character, Double (wrapper classes). |
| Use in Collections | Not allowed directly. | Collections only store objects (autoboxing converts primitives → wrappers). |
| Performance | Faster, less memory overhead. | Slower due to object wrapping (boxing/unboxing). |

**Example:**

int x = 10; // primitive

ArrayList<Integer> list = new ArrayList<>();

list.add(10); // autoboxed into Integer object

**3. Explain linear vs non-linear data structures.**

* **Linear**: Data is arranged sequentially. Every element has a next and previous.  
  ➝ Examples: Array, LinkedList, Stack, Queue.
* **Non-linear**: Data is hierarchical or graph-based.  
  ➝ Examples: Tree, Graph.

*(You said exclude trees/graphs, but interviewers often expect this distinction explained.)*

**4. What is an Abstract Data Type (ADT)? Give examples.**

* **Definition**: An ADT defines **what operations** can be performed, but not **how** they are implemented.
* Examples:
  + **Stack ADT**: push, pop, peek.
  + **Queue ADT**: enqueue, dequeue.
  + **List ADT**: insert, delete, search.

In Java, ADTs are often expressed using **interfaces** (List, Queue, Map), while classes (ArrayList, LinkedList, HashMap) provide the implementation.

**5. Difference between static and dynamic data structures.**

| **Static** | **Dynamic** |
| --- | --- |
| Fixed size (declared at compile time). | Grows/shrinks at runtime. |
| Memory allocated once. | Memory allocated/deallocated as needed. |
| Example: int[] arr = new int[10]; | ArrayList, LinkedList. |

**6. What is time complexity?**

* A measure of how **running time of an algorithm** grows with **input size n**.
* Expressed in Big-O notation (O(n), O(log n)).

**7. What is space complexity?**

* The amount of **extra memory** required by an algorithm (apart from input).
* Includes auxiliary space (temporary variables, recursion stack).

**8. Explain Big-O, Big-Ω, and Big-θ notations.**

* **Big-O (O)**: Upper bound (worst case).
* **Big-Ω (Ω)**: Lower bound (best case).
* **Big-θ (Θ)**: Tight bound (average, exact).

**9. What is amortized analysis?**

* Used when a single operation may be expensive, but over a series of operations the **average cost is small**.
* Example: In ArrayList, resizing (doubling capacity) is expensive O(n), but most insertions are O(1) → **amortized O(1)**.

**10. Difference between best case, average case, and worst case analysis.**

* **Best case**: Minimum time taken (Ω).
* **Average case**: Expected time for random input (Θ).
* **Worst case**: Maximum time taken (O).

**11. What is an array?**

* A **contiguous block of memory** storing elements of the **same type**, accessed via index.

int[] arr = {1, 2, 3, 4};

**12. Advantages and disadvantages of arrays.**

**Advantages**

* Fast random access (O(1)).
* Simple and efficient for fixed-size data.

**Disadvantages**

* Fixed size.
* Insertion/deletion costly (O(n)).
* Wastes memory if not fully used.

**13. Difference between one-dimensional, two-dimensional, and jagged arrays.**

* **1D**: int[] arr = {1, 2, 3};
* **2D**: int[][] arr = new int[3][3]; (matrix-like).
* **Jagged**: Array of arrays with different lengths.
* int[][] jagged = new int[3][];
* jagged[0] = new int[2];
* jagged[1] = new int[4];

**14. How are arrays stored in memory?**

* Stored **contiguously** in heap memory.
* Reference to first element stored in stack.

**15. What is dynamic array?**

* An array that **resizes automatically** when full (like ArrayList).
* Implemented by creating a bigger array and copying elements.

**16. ArrayList vs Vector**

| **Feature** | **ArrayList** | **Vector** |
| --- | --- | --- |
| Thread Safety | Not synchronized. | Synchronized (thread-safe). |
| Performance | Faster (no locking overhead). | Slower (locking). |
| Growth | Increases by 50%. | Doubles capacity. |

**17. Difference between Array and ArrayList in Java.**

* **Array**: Fixed size, stores primitives or objects.
* **ArrayList**: Resizable, stores only objects (autoboxing for primitives).

**18. Difference between LinkedList (Java util) and array-based list.**

| **ArrayList** | **LinkedList** |
| --- | --- |
| Backed by dynamic array. | Backed by doubly linked list. |
| Random access O(1). | Random access O(n). |
| Insert/delete costly (O(n)). | Insert/delete fast at ends (O(1)). |

**19. When would you use an array over a linked list?**

* When **fast random access** is needed.
* When size is known and fixed.
* Example: Storing lookup tables, matrix operations.

**20. Time complexity of insertion, deletion, search in arrays.**

* **Search**: O(n) (linear) or O(log n) (if sorted, binary search).
* **Insertion/Deletion**: O(n) (shifting needed).
* **Access by index**: O(1).

**21. Real-life applications of arrays.**

* Lookup tables (marks of students).
* Image pixel storage.
* Matrix operations.
* Fixed-size buffer storage.

**26. Why does Java not have pointers?**

👉 Java deliberately **does not expose pointers** to developers (unlike C/C++).

**Reasons**:

1. **Security & Safety**:
   * Pointers allow arbitrary memory access → buffer overflow, memory corruption.
   * Java prevents this to make it **robust and secure**.
2. **Portability**:
   * Pointer arithmetic depends on machine architecture (32-bit, 64-bit).
   * Java runs on JVM (WORA: *Write Once, Run Anywhere*). No machine-dependent pointer arithmetic.
3. **Simplicity**:
   * No manual memory management. JVM + Garbage Collector manage memory.

👉 **However:** internally Java *does use references (which behave like safe pointers)*, but you cannot do arithmetic on them.

27.

**Java Generics**

* **Definition**: Generics allow you to write classes, interfaces, and methods with **type parameters**

Without Generics (before Java 5):

ArrayList list = new ArrayList();

list.add("Hello");

list.add(123); // allowed, but unsafe

String s = (String) list.get(1); // runtime ClassCastException

With genriec:

ArrayList<String> list = new ArrayList<>();

list.add("Hello");

// list.add(123); // compile-time error

28. **3. Character Array vs String**

| **Feature** | **Char Array (char[])** | **String** |
| --- | --- | --- |
| Mutability | Mutable (can modify characters). | Immutable (cannot change once created). |
| Storage | Stored in heap (like normal arrays). | Stored in **String Pool** (if literal) or Heap (if created with new). |

29.

**. Difference between Array and Pointer Representation**

* **In C/C++**:
  + Array name acts as a pointer to the first element.
  + You can do pointer arithmetic.

int arr[3] = {1,2,3};

int \*p = arr; // pointer to first element

p++; // moves to next element

* **In Java**:
  + Arrays are **objects in heap**, reference stored in stack.
  + No pointer arithmetic allowed.

int[] arr = {1, 2, 3};

System.out.println(arr[0]); // direct access

👉 So, in Java: **reference = safe pointer without arithmetic**.

30.

**5. Why Collection Framework was introduced?**

**Problem with Arrays (before Collections):**

* Fixed size.
* No built-in sorting/searching.
* Manual resizing.
* Can store only same type (unless using Object[]).

**Solution → Collections Framework**

* Resizable (ArrayList, Vector).
* Supports heterogeneous objects (with Generics).
* Provides algorithms (Collections.sort(), binarySearch()).
* Built-in data structures (HashMap, HashSet).

**If Collections didn’t exist:**

* Developers would need to **implement their own DS** (linked lists, stacks, queues).
* Higher chance of bugs and boilerplate code.

31.

**6. Java String Pool, Heap Memory Creation (with example)**

**Case 1: String Literal**

String s1 = "Hello";

String s2 = "Hello";

* Stored in **String Pool (inside Heap)**.
* Both s1 and s2 point to the **same object**.

**Case 2: Using new keyword**

String s3 = new String("Hello");

* Creates **two objects**:
  + One in String Pool ("Hello")
  + One in Heap (new String object).

32.

**Shallow Copy**

* Copies **only references**, not actual objects.
* Both original and copy refer to same memory.

ArrayList<String> list1 = new ArrayList<>();

list1.add("A");

ArrayList<String> list2 = list1; // shallow copy

list2.add("B");

System.out.println(list1); // [A, B]

**Deep Copy**

* Creates a **new independent object**, copying actual data.

ArrayList<String> list1 = new ArrayList<>();

list1.add("A");

ArrayList<String> list2 = new ArrayList<>(list1); // deep copy

list2.add("B");

System.out.println(list1); // [A]

System.out.println(list2); // [A, B]

33.

**. Difference between == and .equals() for Strings**

| **Operator** | **Checks** | **Example** |
| --- | --- | --- |
| == | Reference equality (same memory address). | "abc" == "abc" → true, "abc" == new String("abc") → false |
| .equals() | Content equality (characters inside). | "abc".equals(new String("abc")) → true |

Eg:

String a = "Hello";

String b = new String("Hello");

System.out.println(a == b); // false (different objects)

System.out.println(a.equals(b)); // true (same content)

34.

**What is String Interning?**

* In Java, **String literals** are stored in a special memory area called the **String Pool** (inside Heap).
* If two string literals have the same value, Java **reuses the same object** instead of creating new ones → this saves memory.
* Interning = ensuring that all strings with the same value **share the same object in the pool**.

Eg:

String s1 = "Java";

String s3 = new String("Java");

System.out.println(s1 == s3); // false (different objects)

String s4 = s3.intern();

 .intern() checks:

* If "Java" is already in the pool → return reference from the pool.
* Else → add it to pool, then return it.

 Since "Java" already exists in the pool, s4 now points to the **pooled object**.

35.

**Why is StringBuilder preferred over String in concatenation?**

* **String** is **immutable** → every time you do concatenation (s = s + "abc"), a **new String object** is created in heap. This wastes memory & reduces performance (O(n²) in loops).
* **StringBuilder** is **mutable** → it modifies the same object without creating new ones. Concatenation becomes efficient (O(n)).

36.

**Difference between mutable and immutable objects in Java**

* **Immutable**: Once created, state cannot change. Example: String, Integer, LocalDate.
* **Mutable**: State (fields) can be changed after creation. Example: StringBuilder, ArrayList.

✅ Example:

String s = "Java";

s.concat(" Programming");

System.out.println(s); // "Java" (not modified, new object created)

StringBuilder sb = new StringBuilder("Java");

sb.append(" Programming");

System.out.println(sb); // "Java Programming" (same object modified)

37.

**How does hashCode() affect Strings in HashMap?**

* HashMap uses **hashCode()** to find the bucket where the key-value pair is stored.
* For String, hashCode() is overridden → it’s calculated from characters.
* If two Strings are **equal (.equals())**, their hashCode() is also the same → required for HashMap consistency.

✅ Example:

HashMap<String, Integer> map = new HashMap<>();

map.put("Java", 1);

map.put("Java", 2); // replaces old value since keys are equal

System.out.println(map.get("Java")); // 2

Behind the scenes:

* "Java".hashCode() → 2301506 (example)
* Both "Java" keys have same hash → go to same bucket → .equals() confirms → value updated.

**38.Difference between substring and subsequence**

* **Substring**: Continuous part of string.  
  Example: "abc" in "abcd".
* **Subsequence**: Characters in the same order, but not necessarily continuous.  
  Example: "acd" is a subsequence of "abcd", but not substring.

**39.8️⃣ Difference between String, StringBuilder, and StringBuffer**

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| **Mutability** | Immutable | Mutable | Mutable |
| **Thread-safety** | Safe (immutable) | ❌ Not thread-safe | ✅ Thread-safe (synchronized) |
| **Performance** | Slow in loops (creates new objects) | Fast | Slower than StringBuilder (due to sync) |
| **Use case** | Constants, keys in maps, read-only text | Fast concatenation in single-threaded apps | Multi-threaded programs where thread safety is needed |

Stack:

**1. What is a Stack?**

* **Definition**:  
  A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle.
* You can only insert (push) or remove (pop) elements from **one end (the top)**.

👉 In Java:

* Legacy: java.util.Stack (class).
* Recommended: ArrayDeque (faster, non-synchronized).

**2. Explain LIFO principle with example.**

* **LIFO = Last In, First Out**
  + The last element inserted is the first one removed.
  + Think of a **stack of plates**: the last plate placed on top is the first one removed.

**Example (Java):**

import java.util.Stack;

public class StackExample {

public static void main(String[] args) {

Stack<String> stack = new Stack<>();

stack.push("A"); // bottom

stack.push("B");

stack.push("C"); // top

System.out.println(stack.pop()); // C (last in → first out)

System.out.println(stack.pop()); // B

System.out.println(stack.pop()); // A

}

}

Output:

C

B

A

**3. Applications of stack in real life.**

* Undo/Redo functionality (MS Word, text editors).
* Browser history (Back button).
* Call stack in programming languages.
* Expression evaluation (e.g., calculators).
* Syntax parsing (compilers, XML/HTML validation).
* Depth-first search (DFS) in graphs.

**4. What are stack overflow and stack underflow?**

* **Stack Overflow**:  
  Happens when you push an element into a stack that is already full (array-based stack with fixed size).
  + In recursion → too many nested function calls cause *stack overflow error*.
* **Stack Underflow**:  
  Happens when you pop an element from an **empty stack**.

**5. How is stack implemented (array vs linked list)?**

**(a) Array-based stack**

* Use a fixed-size array.
* Operations:
  + push() → add at top (increment index).
  + pop() → remove from top (decrement index).
* Limitation: Fixed capacity (overflow possible).

**(b) Linked list-based stack**

* Each node contains data + reference to next.
* Top pointer points to the last pushed node.
* Advantages: Dynamic size (no overflow unless memory full).
* Disadvantage: More memory (extra pointer storage).

**6. Difference between recursion and stack.**

| **Feature** | **Recursion** | **Stack** |
| --- | --- | --- |
| Definition | Function calling itself | Data structure storing elements in LIFO order |
| Relation | Recursion internally uses **call stack** to store function calls | Explicit stack created in memory |
| Example | Factorial recursion (n! = n \* (n-1)!) | Browser back button, Undo |

👉 Interview tip: *Every recursion is implemented using a stack internally by JVM (call stack).*

**7. Applications of stack in parsing and expression evaluation.**

* **Infix to Postfix conversion** (using Shunting-yard algorithm).
* **Postfix evaluation** (Reverse Polish Notation calculators).
* **Balanced parenthesis checking**:
* Input: (a + b) \* (c + d)
* Stack helps ensure brackets match properly.
* Compiler parsing (checking matching tags in XML/HTML).

**8. Limitations of stack.**

* **Fixed size (in array implementation)** → stack overflow possible.
* **Restricted access** → only top accessible, no random access like arrays.
* **Recursive function stack** may cause memory issues for deep recursion.

**9. Real-world examples of stack usage in software.**

* Browser history navigation.
* Undo/Redo in text editors.
* Backtracking algorithms (Maze solving, Sudoku).
* Memory management (function call stack).
* Expression parsing in compilers.

**10. Difference between stack memory and heap memory.**

| **Feature** | **Stack Memory** | **Heap Memory** |
| --- | --- | --- |
| Usage | Stores local variables, method calls (call stack). | Stores objects, arrays, dynamic memory. |
| Access | LIFO (fast, organized). | Random access (slower). |
| Size | Smaller, limited. | Larger, depends on JVM settings. |
| Management | Managed by compiler (automatically allocated & deallocated). | Managed by programmer indirectly (via JVM + Garbage Collector). |
| Error | Stack overflow (too many recursive calls). | OutOfMemoryError (too many objects). |

Time Compelxity:

**1. Basic Stack Operations**

| **Operation** | **Description** | **Time Complexity** |
| --- | --- | --- |
| **push(x)** | Insert an element on top of the stack | **O(1)** |
| **pop()** | Remove the top element | **O(1)** |
| **peek() / top()** | View the top element without removing it | **O(1)** |
| **isEmpty()** | Check if stack is empty | **O(1)** |
| **isFull()** (for array-based fixed stack) | Check if stack is full | **O(1)** |
| **size()** | Return number of elements | **O(1)** |

Queue:

**1. What is a Queue?**

A **queue** is a **linear data structure** that stores elements in an ordered sequence.

* It follows the **FIFO principle (First In, First Out)**.
* Think of a **line at a movie ticket counter** → the first person in line is the first to get served.

In Java, the Queue interface (in **java.util**) provides standard queue functionality, and implementations include LinkedList, ArrayDeque, PriorityQueue.

**🔹 2. FIFO Principle with Example**

**FIFO (First In, First Out):**

* The element inserted first is removed first.
* Example:

Queue: [A, B, C, D]

Enqueue(E) → [A, B, C, D, E]

Dequeue() → removes A → [B, C, D, E]

Real life: **Queue at railway station ticket counter**.

**🔹 3. Difference Between Stack and Queue**

| **Feature** | **Stack** | **Queue** |
| --- | --- | --- |
| Principle | **LIFO** (Last In First Out) | **FIFO** (First In First Out) |
| Insertion | Push at **top** | Enqueue at **rear** |
| Deletion | Pop from **top** | Dequeue from **front** |
| Example | Plate stack | Waiting line |

**🔹 4. What is a Circular Queue?**

* A **queue that connects the end back to the front** (like a circle).
* Helps **reuse empty spaces** in an array-based queue after deletions.
* Example: If rear reaches end of array, it wraps around to index 0 (if empty slots exist).

**🔹 5. What is a Double-Ended Queue (Deque)?**

* A **deque (pronounced “deck”)** allows **insertion and deletion from both ends**.
* More flexible than queue/stack.
* Java: ArrayDeque class implements it.
* Example:
  + Insert left → [X, A, B]
  + Insert right → [A, B, X]

**🔹 6. What is a Priority Queue?**

* A **special queue** where elements are served based on **priority**, not insertion order.
* Highest (or lowest) priority element is dequeued first.
* Implemented using **Heap** internally in Java.
* Example: In a hospital emergency room, the most critical patient is served first, not the one who arrived first.

**🔹 7. Applications of Queues**

* **CPU scheduling** (Round Robin → Circular Queue)
* **Printer job scheduling**
* **Call center systems** (first caller gets served first)
* **Breadth First Search (BFS)** in graphs
* **I/O buffering** (keyboard input, streaming data)
* **Message queues** in distributed systems

**🔹 8. Queue Implementation**

**(a) Array-based Queue**

* Use fixed-size array.
* **Enqueue** at rear, **Dequeue** from front.
* Limitation: Wastage of space if not circular.
* Operations: O(1) (except resizing).

**(b) Linked List-based Queue**

* Each node has data + next.
* **Enqueue** at tail, **Dequeue** at head.
* No fixed size, no overflow (until memory full).
* Operations: O(1).

**8.Recursion vs iteration Definition**

| **Feature** | **Recursion** | **Iteration** |
| --- | --- | --- |
| Definition | A function **calls itself** directly or indirectly to solve a problem. | A problem is solved using **loops** (for, while, do-while). |
| Approach | **Divide and Conquer**: Break problem into smaller sub-problems. | **Step-by-step repetition** until condition is met. |
| Memory | Uses **call stack** for each function call. | Uses **constant memory** (loop variables). |
| Termination | Must have a **base condition** to stop recursion. | Must have a **loop exit condition**. |