**COLLECTION FRAMEWORK:**

**🔹 1. What is the Collection Framework in Java?**

**Answer:**  
It’s a unified architecture to store, retrieve, and manipulate groups of objects. It includes interfaces like List, Set, Map, and classes like ArrayList, HashSet, HashMap.

**🔹 2. What is the root interface of Collection Framework?**

**Answer:**

* java.util.Collection is the root interface for most.
* Map does **not** extend Collection.

**🔹 3. What is the difference between List, Set, and Map?**

| **Feature** | **List** | **Set** | **Map** |
| --- | --- | --- | --- |
| Order | Maintains order | No guaranteed order | Key-value pair |
| Duplicates | Allowed | Not allowed | Keys unique, values can repeat |
| Interface | List | Set | Map |

**🔹 4. Difference between ArrayList and LinkedList**

| **Feature** | **ArrayList** | **LinkedList** |
| --- | --- | --- |
| Data Structure | Dynamic array | Doubly linked list |
| Access Time | Faster (index based) | Slower (traverse) |
| Insert/Delete | Slow (resize/shift) | Fast (just pointers) |

**🔹 5. Difference between HashSet and TreeSet**

| **Feature** | **HashSet** | **TreeSet** |
| --- | --- | --- |
| Order | No order | Sorted (natural/custom) |
| Performance | Faster | Slower |
| Nulls | Allows one null | Doesn't allow nulls |

**🔹 6. Difference between HashMap and TreeMap**

| **Feature** | **HashMap** | **TreeMap** |
| --- | --- | --- |
| Order | No order | Sorted by key |
| Null Key | Allows one null key | Doesn’t allow null key |
| Performance | Faster | Slower |

**🔹 7. What is the difference between Iterator and ListIterator?**

| **Feature** | **Iterator** | **ListIterator** |
| --- | --- | --- |
| Direction | Forward only | Both forward & backward |
| Used for | Any Collection | Only List |
| Methods | hasNext(), next() | add(), hasPrevious() etc. |

**🔹 8. What is fail-fast vs fail-safe iterator?**

| **Type** | **Fail-Fast** | **Fail-Safe** |
| --- | --- | --- |
| Behavior | Throws ConcurrentModificationException if modified during iteration | Works on cloned copy |
| Example | ArrayList, HashSet | CopyOnWriteArrayList, ConcurrentHashMap |

**🔹 9. What is the difference between HashMap and Hashtable?**

| **Feature** | **HashMap** | **Hashtable** |
| --- | --- | --- |
| Thread-safe | No | Yes (synchronized) |
| Null allowed | 1 null key, multiple null values | None allowed |
| Performance | Better (no sync) | Slower (sync overhead) |

**🔹 10. When to use which collection?**

| **Use Case** | **Best Choice** |
| --- | --- |
| Random access | ArrayList |
| Frequent insert/delete | LinkedList |
| Unique elements, fast search | HashSet / HashMap |
| Sorted data | TreeSet / TreeMap |
| Thread-safe map | ConcurrentHashMap |

**🔹 11. How does HashMap work internally?**

**Answer:**

* Uses array of buckets.
* Each bucket stores a linked list (or tree after Java 8).
* Key's hashcode is used to find bucket index.
* If same hash, equals() used to resolve collision.

**🔹 12. What are comparable and comparator interfaces?**

**Comparable:**

* Used for natural ordering (String, Integer).

public int compareTo(Object o);

**Comparator:**

* Custom sort logic.

public int compare(Object o1, Object o2);

**🔹 13. Difference between Collection and Collections?**

| **Collection** | **Collections** |
| --- | --- |
| Interface | Utility class |
| Part of hierarchy | Contains static methods |
| Example: List, Set | Example: Collections.sort() |

**🔹 14. How to synchronize a collection?**

List<String> list = Collections.synchronizedList(new ArrayList<>());

Or use thread-safe alternatives like CopyOnWriteArrayList.

**🔹 15. What is EnumSet and EnumMap?**

* EnumSet – Efficient Set implementation for enums only.
* EnumMap – Map with enum keys. More efficient than HashMap.

**Advanced Collection Interview Questions:**

**🔸 16. Can a HashMap have duplicate keys or values?**

**Answer:**

* Duplicate **keys**: ❌ Not allowed (latest value replaces previous one).
* Duplicate **values**: ✅ Allowed.

**🔸 17. What happens if you override equals() but not hashCode() in a key class used in HashMap?**

**Answer:**

* Violates the contract: Equal objects must have equal hashCodes.
* Causes **unexpected behavior** in HashMap — may not retrieve correct value.

**🔸 18. Which collection classes are thread-safe?**

* Vector, Stack, Hashtable (legacy, synchronized)
* ConcurrentHashMap, CopyOnWriteArrayList, ConcurrentSkipListMap (modern, better performance)

**🔸 19. How is ConcurrentHashMap better than Hashtable?**

**Answer:**

* Uses **bucket-level locking** (segments or node-level locking in Java 8+).
* Allows **concurrent read/write** without blocking entire map.

**🔸 20. Difference between peek(), poll(), and remove() in Queue?**

| **Method** | **Returns** | **Removes?** | **Exception?** |
| --- | --- | --- | --- |
| peek() | Head | ❌ | ❌ (null if empty) |
| poll() | Head | ✅ | ❌ (null if empty) |
| remove() | Head | ✅ | ✅ (NoSuchElementException if empty) |

**🔸 21. What is identity-based hashing in Java?**

**Answer:**

* Done via IdentityHashMap , uses == instead of .equals() to compare keys.

**🔸 22. What are WeakHashMap and its use cases?**

**Answer:**

* Keys are weakly referenced.
* If a key is no longer referenced elsewhere, it's GC'd.
* Good for **caching** or **memory-sensitive** maps.

**🔸 23. How to sort a Map by keys or values?**

map.entrySet().stream()

.sorted(Map.Entry.comparingByKey()) // or comparingByValue()

.forEach(System.out::println);

**🔸 24. What is NavigableSet and NavigableMap?**

**Answer:**

* Extends SortedSet/SortedMap with navigation methods:
  + lower(), floor(), ceiling(), higher()
  + Useful for **range-based** operations.

**🔸 25. Difference between PriorityQueue and TreeSet?**

| **Feature** | **PriorityQueue** | **TreeSet** |
| --- | --- | --- |
| Duplicates | Allowed | Not allowed |
| Sorting | Heap-based (priority) | Sorted based on comparator |
| Access order | Head only (poll, peek) | Iteration in sorted order |

**🔸 26. Real-world scenario:**

**"How would you implement a leaderboard system?"**

**Answer:**

* Use TreeMap<Integer, List<String>>
  + Score as key (sorted), user list as value
* For descending order: new TreeMap<>(Collections.reverseOrder())

**🔸 27. Real-world scenario:**

**"You need to cache users and invalidate them after 10 mins – what will you use?"**

**Answer:**

* Use LinkedHashMap with access order + override removeEldestEntry()  
  OR
* Use Guava Cache or Caffeine for TTL-based eviction  
  OR
* Use WeakHashMap if you rely on GC behavior

**Java 8 Features :**

**🔹 1. What are the main features introduced in Java 8?**

**Answer:**

* **Lambda Expressions**
* **Functional Interfaces**
* **Stream API**
* **Default & Static Methods in Interfaces**
* **Method References**
* **Optional Class**
* **Date and Time API (java.time)**
* **Nashorn JavaScript Engine**
* **Collectors, Predicate, Function, Consumer (Functional Interfaces)**

**🔹 2. What is a Lambda Expression?**

**Answer:**

* Lambda is a way to represent anonymous methods (functional programming style).
* Syntax: (parameters) -> expression

**Example:**

List<String> names = Arrays.asList("Raj", "Ravi", "Rahul");

names.forEach(name -> System.out.println(name));

**🔹 3. What is a Functional Interface?**

**Answer:**

* An interface with only **one abstract method**.
* Can have default and static methods.
* Annotated with @FunctionalInterface.

**Example:**

@FunctionalInterface

interface MyFunc {

void show();

}

**🔹 4. What is Stream API in Java 8?**

**Answer:**

* Used to **process collections** in a **declarative** and **parallel** way.
* Supports **map, filter, reduce, collect, sorted, etc.**

**Example:**

List<String> names = Arrays.asList("Alex", "Brian", "Charles");

names.stream().filter(name -> name.startsWith("A")).forEach(System.out::println);

**🔹 5. Difference between Stream map() and flatMap()?**

| **Method** | **Description** |
| --- | --- |
| map() | Transform each element |
| flatMap() | Flatten streams of collections |

**🔹 6. What is Optional in Java 8?**

**Answer:**

* A container for a **value that may or may not be present**.
* Helps avoid **NullPointerException**.

**Example:**

Optional<String> name = Optional.of("John");

name.ifPresent(System.out::println);

**🔹 7. What are Default and Static methods in interfaces?**

**Answer:**

* **Default methods**: Have a body and can be inherited.
* **Static methods**: Can be called from the interface directly.

interface MyInterface {

default void sayHello() { System.out.println("Hello"); }

static void sayStatic() { System.out.println("Static method"); }

}

**🔹 8. What is Method Reference in Java 8?**

**Answer:**

* Shorter syntax for calling a method by referring to it by name.

**Syntax:**

ClassName::methodName

**Example:**

names.forEach(System.out::println);

**🔹 9. How is Java 8 Date and Time API better than Date/Calendar?**

**Answer:**

* Immutable and thread-safe
* More readable and fluent
* Example: LocalDate, LocalTime, ZonedDateTime

LocalDate date = LocalDate.now();

**🔹 10. What is the use of Collectors.toList() in streams?**

**Answer:**

* Used to collect stream elements into a List.

List<String> result = stream.filter(...).collect(Collectors.toList());

**🔹 11. What is the difference between intermediate and terminal operations in Stream API?**

| **Type** | **Examples** |
| --- | --- |
| Intermediate | map(), filter(), sorted() |
| Terminal | collect(), forEach(), count() |

**🔹 12. What are some built-in functional interfaces in Java 8?**

* Predicate<T> – returns boolean
* Function<T, R> – returns result
* Consumer<T> – returns void
* Supplier<T> – returns a value

**🔹 13. Can you parallelize stream operations?**

**Answer:** Yes. Use .parallelStream() to improve performance for large data sets (only if operations are stateless and non-blocking).

list.parallelStream().forEach(System.out::println);

**🔹 14. What is the difference between Iterator and Stream in Java 8?**

| **Feature** | **Iterator** | **Stream** |
| --- | --- | --- |
| Traversal | External | Internal |
| Parallelization | ❌ No | ✅ Yes |
| Lazy Eval | ❌ No | ✅ Yes (lazy ops) |
|  |  |  |

**Advanced/Edge Topics in Java 8 Interview Q&A**

**🔹 1. How do you handle exceptions in lambda expressions?**

**Answer:** Lambdas don’t allow checked exceptions directly, so you must handle them inside the lambda or write a wrapper.

list.forEach(item -> {

try {

methodThatThrowsCheckedException(item);

} catch (IOException e) {

e.printStackTrace();

}

});

**🔹 2. How is Java 8 functional programming style different from OOP?**

| **OOP** | **Functional Programming** |
| --- | --- |
| Focus on objects/state | Focus on pure functions/immutability |
| Methods have side effects | Tries to avoid side effects |
| Imperative style | Declarative style (via Stream API) |

**🔹 3. What is the difference between findFirst() and findAny()?**

* findFirst() – returns the **first** element in the stream.
* findAny() – returns **any** element, may be faster with parallel streams.

**🔹 4. What is peek() used for in Stream API?**

* Used for **debugging**, it lets you see what’s happening in the stream.

stream.peek(System.out::println).filter(...);

**🔹 5. Difference between map() and filter() in streams?**

| **map()** | **filter()** |
| --- | --- |
| Transforms data | Filters data (boolean test) |
| Input = A, Output = B | Input = A, Output = A or skip |

**🔹 6. What is the purpose of Optional.orElse() and Optional.orElseGet()?**

* orElse() – returns a default value.
* orElseGet() – uses a **Supplier**, and is **lazy-evaluated**.

String name = optionalName.orElse("Default");

String lazyName = optionalName.orElseGet(() -> "Generated");

**🔹 7. What’s the difference between limit() and skip()?**

* limit(n) – keeps only first n elements.
* skip(n) – skips first n elements.

**🔹 8. What are primitive streams in Java 8?**

Java 8 provides **specialized streams** for primitives:

* IntStream
* LongStream
* DoubleStream

Used to avoid boxing/unboxing overhead.

**🔹 9. Can streams be reused?**

**No**, streams can be consumed only once. Reusing a stream throws an IllegalStateException.

**🔹 10. What is the difference between Collectors.toList() and Collectors.toSet()?**

* toList() – maintains order, allows duplicates.
* toSet() – no duplicates, no guaranteed order.

**🔹 11. What is reduce() in Stream API?**

Used to **accumulate** or combine elements of a stream into a single result.

int sum = list.stream().reduce(0, (a, b) -> a + b);

**🔹 12. Difference between forEach() and forEachOrdered()?**

| **forEach()** | **forEachOrdered()** |
| --- | --- |
| No guarantee of order | Preserves encounter order |
| Faster (parallel) | Slower (parallel) |

**🧩 Java 17 Features (LTS - September 2021):**

Java 17 is a **Long-Term Support (LTS)** version and brought many stable and finalized features from earlier preview releases.

**✅ Key Java 17 Features:**

| **Feature** | **Description** |
| --- | --- |
| ✅ **Sealed Classes** | Restrict which classes can extend or implement a class/interface. |
| ✅ **Pattern Matching for instanceof** | Cleaner type casting after instanceof checks. |
| ✅ **Switch Enhancements (Preview)** | Pattern matching in switch (preview). |
| ✅ **Records** (finalized in Java 16) | Immutable data carriers, reduce boilerplate. |
| ✅ **Text Blocks** | Multiline strings with triple quotes ("""). |
| ✅ **JEP 356: Enhanced Pseudo-Random Number Generators** | More flexible and pluggable PRNGs. |
| ✅ **JEP 382: New macOS rendering pipeline** | Uses Metal instead of the deprecated OpenGL. |

**🔍 Example: Pattern Matching for instanceof**

if (obj instanceof String s) {

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

}

**🔮 Java 21 Features (LTS - September 2023):**

Java 21 is also an **LTS version**, and includes some powerful new **language, performance, and tooling enhancements**.

**✅ Key Java 21 Features:**

| **Feature** | **Description** |
| --- | --- |
| ✅ **String Templates (Preview)** | Interpolate expressions inside strings directly. |
| ✅ **Record Patterns (Preview)** | Deconstruct Record objects in pattern matching. |
| ✅ **Virtual Threads** | Lightweight threads from Project Loom, helps scale concurrent apps. |
| ✅ **Sequenced Collections** | New interfaces: SequencedCollection, SequencedSet, SequencedMap. |
| ✅ **Scoped Values** | Efficient thread-local-like mechanism for virtual threads. |
| ✅ **Pattern Matching for switch (Finalized)** | Cleaner switch-case logic using types and conditions. |
| ✅ **Unnamed Classes & Instance Main Methods (Preview)** | Simplifies writing small programs without boilerplate. |
| ✅ **JEP 445: Unnamed Patterns and Variables** | Use \_ as a discard symbol in patterns. |

**🔍 Example: Virtual Threads (Java 21)**

Runnable task = () -> System.out.println("Running in thread " + Thread.currentThread());

Thread t = Thread.startVirtualThread(task);

**🔍 Example: String Templates (Java 21 Preview)**

String name = "Alice";

String message = STR."Hello, \{name}!";

**🔄 Summary: Java 17 vs Java 21**

| **Feature Area** | **Java 17** | **Java 21** |
| --- | --- | --- |
| LTS? | ✅ Yes | ✅ Yes |
| Pattern Matching | instanceof | switch, record patterns |
| Records | Finalized | Extended usage in patterns |
| Virtual Threads | ❌ Not present | ✅ Major feature (Project Loom) |
| String Templates | ❌ | ✅ (Preview) |
| Performance | Better GC, faster JIT | + Virtual Threads, Scoped Values |
| Collections | Standard | + Sequenced Collections |
| Ease of Learning | Traditional boilerplate | Unnamed classes/main methods |