Java 8 Stream Interview Programs

Version 0.1

**Document Overview**

This document is intended to specify the interview processes and actual scenarios asked at the time of interview.

**Purpose**

The purpose of this document is to describe what needs to learn the concepts will describe and learn prepare for java interview.

**Scope**

The information contained in this document provides the detailed functional analysis for the interview process.

**Version Log**

| **Date** | **Author** | **Version** | **Change Description** |
| --- | --- | --- | --- |
| 04/17/2025 | Srikanth Mamillapalli | 0.1 | Initial Draft |

Table of Contents

[1. Core Java Important Concepts for Interview 5](#_Toc198153662)

[1.1 Java 8 Features 5](#_Toc198153663)

[1.2 Java8 Map new Features 5](#_Toc198153664)

[1.3 Core DatStructures in Java Collection Framework 6](#_Toc198153665)

[1.4 Important Implementations & Their Key Features 7](#_Toc198153666)

[1.5 Aggregate (Reduction) Functions in Java 8 Streams 7](#_Toc198153667)

[1.6 Parallel streams 8](#_Toc198153668)

[1.7 CompletableFuture 8](#_Toc198153669)

[1.8 Optional 8](#_Toc198153670)

[1.9 Lambda Expression 9](#_Toc198153671)

[1.10 Functional Interfaces (SAM) 10](#_Toc198153672)

[1.11 Java Map works internal process 11](#_Toc198153673)

[1.12 HashMap uses a combination of: 11](#_Toc198153674)

[1.13 Hashing and Index Calculation 12](#_Toc198153675)

[1.14 ConcurrentHashMap works internally 13](#_Toc198153676)

[1.15 Internal Process (Step-by-Step) 13](#_Toc198153677)

[1.16 Collection and Internal uses 14](#_Toc198153678)

[1.17 What is the difference between HashMap and ConcurrentHashMap? 15](#_Toc198153679)

[1.18 Thread Safety 16](#_Toc198153680)

[1.19 Comparable vs comparator 16](#_Toc198153681)

[1.17.1 Comparable Interface: 16](#_Toc198153682)

[1.17.2 Comparator Interface: 16](#_Toc198153683)

[1.20 Java 8 Enhancements 17](#_Toc198153684)

[1.21 Techniques to Make Code Thread-Safe 18](#_Toc198153685)

[1.22 Custom LinkedList Implementation 19](#_Toc198153686)

[1.23 What is callable and runnable interfaces 21](#_Toc198153687)

[1.24 Key Characteristics of Static Methods in Interfaces 22](#_Toc198153688)

[1.25 Key Characteristics of Default Methods 23](#_Toc198153689)

[1.26 Method References 23](#_Toc198153690)

[1.27 Java11 features 24](#_Toc198153691)

[1.28 Java21 features 25](#_Toc198153692)

[1.29 Virtual Thread 25](#_Toc198153693)

[1.30 Key Features 25](#_Toc198153694)

[2. DataBase 26](#_Toc198153695)

[2.1 How to increase the performance or tune the queries 26](#_Toc198153696)

[2.2 What is the cursor and refcursor in pl/sql 27](#_Toc198153697)

[2.2.1 Cursor : 27](#_Toc198153698)

[2.2.2 Ref Cursor : Reference Cursor 28](#_Toc198153699)

[2.2.3 Example returning a refcursor: 28](#_Toc198153700)

[2.3 Triggers in database: 29](#_Toc198153701)

[2.4 IN,OUT,IN\_OUT parameters in database stored procedure 29](#_Toc198153702)

[2.5 Raise an exception in stored procedure 30](#_Toc198153703)

[2.6 How to handle errors/exception in stored procedure 30](#_Toc198153704)

[2.7 Different types of data types 31](#_Toc198153705)

[2.8 Different types of joins 31](#_Toc198153706)

[2.9 execute a stored procedure from java/springboot/jpa 31](#_Toc198153707)

[2.10 Different queries 32](#_Toc198153708)

[2.11 CTE common table expressions 33](#_Toc198153709)

[2.12 SnowFlake and Databrics 33](#_Toc198153710)

[2.13 JPA Repository and CURD Repository 34](#_Toc198153711)

[2.14 Pagination logic 35](#_Toc198153712)

[2.15 Transaction 36](#_Toc198153713)

[3. Core Java 8 Stream Important programs for Interview 38](#_Toc198153714)

[3.1 CountPairs for given set of numbers 38](#_Toc198153715)

[3.2 CharacterCountUsingHashMap 38](#_Toc198153716)

[3.3 Find the pair with the sum=10 for a given numbers 39](#_Toc198153717)

[3.4 RepeatingCharacters for a given String 39](#_Toc198153718)

[3.5 Concat or merge two maps 39](#_Toc198153719)

[3.6 StreamCollectJoiningExample 40](#_Toc198153720)

[3.7 Stream Collect toList example 40](#_Toc198153721)

[3.8 StreamCollectToMapExample 40](#_Toc198153722)

[3.9 StreamCollect toSet Example 41](#_Toc198153723)

[3.10 Find max number from emptyList 41](#_Toc198153724)

[3.11 Find max number from the given list 41](#_Toc198153725)

[3.12 How to get the index of the array element 42](#_Toc198153726)

[3.13 list of employees filter by dept -> tech filter by salry >50000 sort in descending order by name collect data LinkedList using java8 streams 43](#_Toc198153727)

[3.14 find non repeatitive first character using java8 43](#_Toc198153728)

[3.15 find the name and length of a string using java8 43](#_Toc198153729)

[3.16 Produce deadlock code 43](#_Toc198153730)

[3.17 FindArrayIndexElement 44](#_Toc198153731)

[3.18 FilteredSortedEmployees 44](#_Toc198153732)

[3.19 RepeatingCharacters 45](#_Toc198153733)

[3.20 IntegerArrayDemo findPairs 45](#_Toc198153734)

[3.21 Character occurrences in a map 46](#_Toc198153735)

[3.22 CountPairs in an array 46](#_Toc198153736)

[3.23 StreamMaxStringExample 47](#_Toc198153737)

[3.24 StreamReduceWithIdentityExample 47](#_Toc198153738)

[3.25 findLargestString 47](#_Toc198153739)

[3.26 StreamForEachParallelExample 48](#_Toc198153740)

[3.27 ConcatinationMapsDemo 48](#_Toc198153741)

[3.28 Longest substring without repeating characters 49](#_Toc198153742)

# Core Java Important Concepts for Interview

### java different version features

|  |  |  |
| --- | --- | --- |
| Java8 | Java11 | Java21 |
| Lambda Expressions | Local-Variable Syntax for  Lambda Parameters | Record Patterns |
| Functional Interfaces | New String Methods | Pattern Matching for switch |
| Method References | New File Methods | Virtual Threads |
| Default Methods | HTTP Client (Standardized) | Sequenced Collections |
| Static Methods in Interfaces | Collection Enhancements | String Templates |
| Stream API | Launch Single-File Programs | Incubator Features |
| Date and Time API |  | Scoped Values |
| Optional Class |  | Unnamed Patterns and Variables |
| Nashorn JavaScript Engine |  | Unnamed Classes |
| Parallel Streams |  | Instance Main Methods |
| CompletableFuture |  | Foreign Function & Memory API |
| Collectors Utility |  | Structured Concurrency |

### Java 8 Features

|  |  |  |
| --- | --- | --- |
| Java8 | | |
| Features | Streams | Functional Interfaces |
| Lambda Expressions | **intermediate operations** | Function<T, R> |
| Functional Interfaces | Filter(Predicate) | BiFunction<T, U, R> |
| Method References | Map(Function) | Consumer<T> |
| Default Methods in Interfaces | flatMap Function<T, Stream<R>> | BiConsumer<T, U> |
| Static Methods in Interfaces | Distinct uses Object.equals() and hashCode() | Supplier<T> |
| Stream API | sorted() Comparator<T> | Predicate<T> |
| Date and Time API (java.time) | limit(n) | BiPredicate<T, U> |
| Optional Class | skip(n) | UnaryOperator<T> |
| Nashorn JavaScript Engine | Peek Consumer<T> | BinaryOperator<T> |
| Parallel Streams | **Terminal Operations** | **Map new Features** |
| CompletableFuture | collect(Collectors) | forEach |
| Collectors Utility | forEach(Consumer) | getOrDefault |
|  | count() | putIfAbsent |
|  | anyMatch() Predicate<T> | computeIfAbsent |
|  | allMatch() Predicate<T> | computeIfPresent |
|  | noneMatch()Predicate<T> | compute |
|  | findFirst() Retrieves optional result | merge |
|  | findAny() Retrieves optional result | Replace |
|  | reduce() BinaryOperator<T> | remove |

### Important Implementations & Their Key Features

| **Class** | **Description** |
| --- | --- |
| **ArrayList** | Resizable array, fast random access, slower insert/remove in middle |
| **LinkedList** | Doubly linked list, fast insert/remove, slower random access |
| **HashSet** | Uses HashMap internally, fast lookup, no guaranteed order |
| **LinkedHashSet** | Maintains insertion order, still no duplicates |
| **TreeSet** | Sorted set based on Red-Black tree, elements must be Comparable or use Comparator |
| **HashMap** | Most commonly used map, allows null keys/values, no order |
| **LinkedHashMap** | Maintains insertion order or access order |
| **TreeMap** | Sorted map, keys must be Comparable or use Comparator |
| **PriorityQueue** | Elements ordered according to their natural ordering or Comparator |
| **ArrayDeque** | Efficient double-ended queue, better than Stack and LinkedList for stack/queue usage |

### Aggregate (Reduction) Functions in Java 8 Streams

**1. count()**

Returns the number of elements in the stream.

long count = list.stream().count();

**2. sum() (via mapToInt(), mapToLong(), or mapToDouble())**

Calculates the sum of numeric values.

int sum = list.stream().mapToInt(Integer::intValue).sum();

**3. average()**

Returns an OptionalDouble representing the average of the elements.

OptionalDouble avg = list.stream().mapToInt(Integer::intValue).average();

**4. min() / max()**

Returns the minimum or maximum element in the stream.

Optional<Integer> min = list.stream().min(Integer::compareTo);

Optional<Integer> max = list.stream().max(Integer::compareTo);

**5. reduce()**

General-purpose reduction method. Can be used for sum, product, concatenation, etc.

Optional<Integer> sum = list.stream().reduce((a, b) -> a + b);

Or with an identity:

int sum = list.stream().reduce(0, Integer::sum);

**6. collect()**

A powerful method for custom aggregation. Common collectors:

* Collectors.toList() / toSet()
* Collectors.joining()
* Collectors.averagingInt()
* Collectors.summingInt()
* Collectors.summarizingInt() → gives count, sum, min, average, max in one shot

int total = list.stream().collect(Collectors.summingInt(Integer::intValue));

Or get full stats:

IntSummaryStatistics stats = list.stream().collect(Collectors.summarizingInt(Integer::intValue));

**7. groupingBy()**

Group elements by a classifier function and apply aggregation.

Map<String, Long> groupCount = list.stream()

.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

Predicate<Integer> isEven = number -> number % 2 == 0;

System.out.println("Is 10 even? " + isEven.test(10));

Function<String, Integer> stringLength = str -> str.length();

System.out.println("Length of 'Hello': " + stringLength.apply("Hello"));

### Parallel streams

It make use of the **fork-join framework** and its common pool of worker threads.

### CompletableFuture

It is a class in Java (java.util.concurrent) that allows you to write **asynchronous**, **non-blocking** code. It helps you run tasks in the background and then continue processing once they’re complete.

### Optional

The Optional class in Java 8 is a **container object** used to contain not-null objects. Optional is part of the java.util package and was introduced to reduce the risk of NullPointerException and provide a better way to handle optional values.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

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

name.ifPresent(n -> System.out.println(n.toUpperCase()));

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

System.out.println(result);

Optional<String> emptyName = Optional.empty();

String result2 = emptyName.orElse("Default");

System.out.println(result2);

When to Use Optional

* As a return type when the result **might be absent**.
* To **replace null checks**.
* To encourage **functional-style programming**

### Lambda Expression

Lambda expressions provide a concise way to implement the abstract method of a functional interface.

Lambda expressions let you write shorter, cleaner, and more expressive code, especially for functional interfaces.

They are extensively used with **Streams**, **Collections**, and **multithreading**.

### Functional Interfaces (SAM)

Functional interface is an interface that contains exactly one abstract method.

These interfaces are used primarily with lambda expressions and method references, enabling you to write cleaner and more concise code.

functional interfaces are interfaces with a single abstract method (SAM), and they can be used as the target for lambda expressions or method references.

|  |  |
| --- | --- |
| Functional Interfaces | Example |
| Function<T, R> | Function<String, Integer> lengthFunction = str -> str.length();  System.out.println(lengthFunction.**apply**("Sree")); |
| BiFunction<T, U, R> | BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;  System.out.println(sum.**apply**(5, 3)); |
| Consumer<T> | Consumer<String> consumer = name -> System.out.println("Hello, " + name);  consumer.**accept**("Alice"); |
| BiConsumer<T, U> | BiConsumer<String, Integer> print = (name, age) ->  System.out.println(name + " is " + age + " years old");  print.**accept**("Bob", 30); |
| Supplier<T> | Supplier<String> supplier = () -> "Hello from Supplier!";  System.out.println(supplier.**get**()); |
| Predicate<T> | Predicate<String> isLongerThan5 = s -> s.length() > 5;  System.out.println(isLongerThan5.**test**("Hello")); |
| BiPredicate<T, U> | BiPredicate<String, String> isEqual = (a, b) -> a.equalsIgnoreCase(b);  System.out.println(isEqual.test("Java", "java")); |
| UnaryOperator<T> | UnaryOperator<String> toUpperCase = s -> s.toUpperCase();  System.out.println(toUpperCase.**apply**("hello")); |
| BinaryOperator<T> | BinaryOperator<Integer> multiply = (a, b) -> a \* b;  System.out.println(multiply.**apply**(4, 5)); |

Annotated with **@FunctionalInterface** – this is optional, but it helps catch errors during

compile-time.

### Java Map works internal process

**HashMap works internally**

1. HashMap works on the priniciple of Hashing technology.
2. A HashMap stores **key-value pairs** and allows constant-time performance for get() and put() operations
3. To generate hash index it used hash code and it returns integer number.

| **Concept** | **Description** |
| --- | --- |
| Bucket | A location in an internal array where entries are stored. |
| Hash Function | Used to convert the key into an array index (bucket index). |
| Collision | When two keys hash to the same bucket. Handled using a linked list or a balanced tree (Java 8+). |
| Load Factor | Determines when the HashMap should resize. Default is 0.75. |
| Capacity | The size of the internal array (must be a power of 2). |

1. **Hashing the Key** int hash = hash(key.hashCode());

The key's hashCode() is passed through a supplemental hash function to reduce collisions.

1. **Calculating the Index**

int index = hash & (n - 1); Where n is the length of the table (must be a power of 2).

1. **Check for Collision**

* If the calculated bucket is empty, place the entry there.
* If not, iterate through the bucket:
* If the key already exists (via equals()), update the value.
* Otherwise, add the new node to the linked list (or tree if large enough).

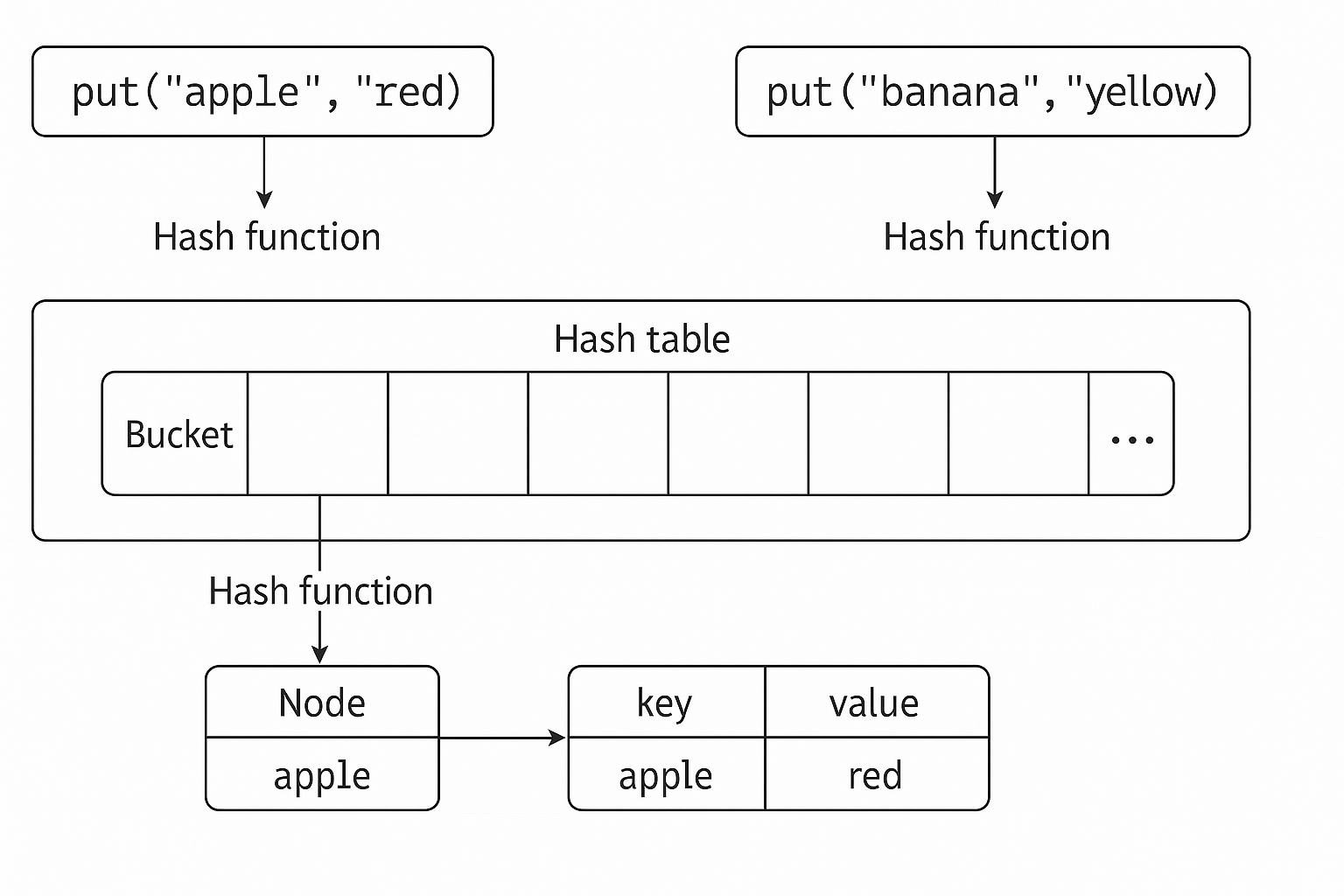
1. If the number of entries in a bucket exceeds a threshold (default 8), the bucket is converted from a **LinkedList to a Red-Black Tree** for faster lookups (O(log n)).

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

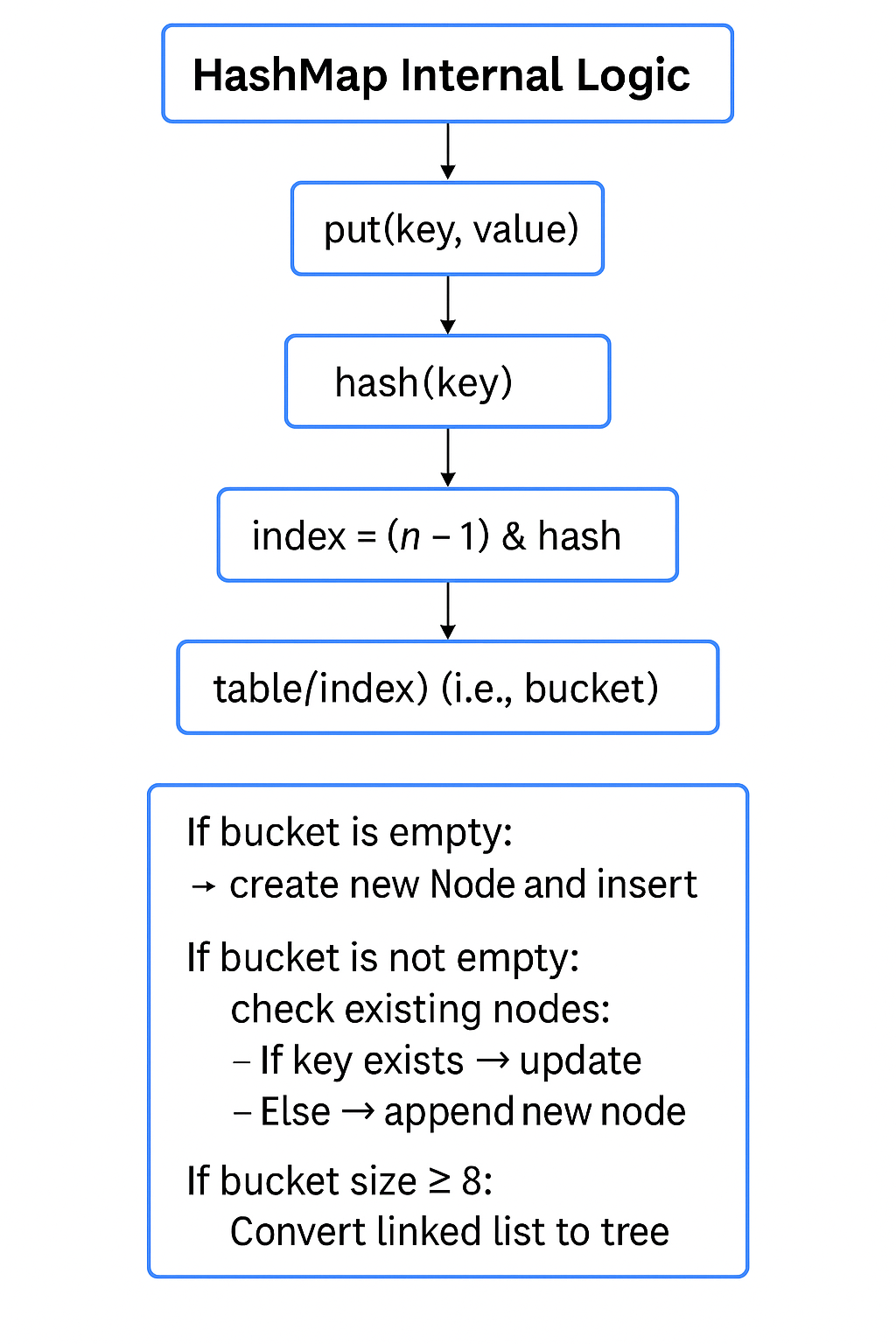
map.put("apple", "red");

map.put("banana", "yellow");

* + apple.hashCode() → hash value
  + Index is computed from hash
  + Bucket is checked
  + If no collision → stored; if collision → linked list/tree updated



### Java Map works internal process image



| **Map Type** | **Internal Structure** | **Ordered?** | **Sorted?** |
| --- | --- | --- | --- |
| HashMap | Array + LinkedList / Tree | ❌ | ❌ |
| LinkedHashMap | Same as HashMap + Linked List for order | ✅ | ❌ |
| TreeMap | Red-Black Tree | ✅ | ✅ (Sorted by natural/comparator) |
| ConcurrentHashMap | Segmented lock or CAS for concurrency | ❌ | ❌ |

### ConcurrentHashMap works internally

* It uses a single **array of Nodes** (like HashMap).
* Each bin (bucket) can be:
  + A linked list (Node<K,V>)
* A tree (red-black tree) when the number of nodes exceeds a threshold (like HashMap)
* Synchronization happens at the **bucket level**, not segment level.

**Java 8 redesigned ConcurrentHashMap:**

* **No more segment locking** (unlike Java 7)
* Uses a **Node[] table**, similar to HashMap
* Each bucket contains a **linked list or tree** of Node<K,V> entries
* **Fine-grained locking via CAS + synchronized blocks**

A screenshot of a computer

AI-generated content may be incorrect.

### Collection Internal use collection

A screenshot of a computer

AI-generated content may be incorrect.

### Collection Internal Data Structure

A screenshot of a computer

AI-generated content may be incorrect.

### What is the difference between HashMap and ConcurrentHashMap?

* **HashMap** is not thread-safe and may produce inconsistent results in multithreaded environments.
* **ConcurrentHashMap** allows concurrent access with thread safety using internal segment locking.

### Thread Safety

A **thread-safe** class or method ensures that **shared data is accessed and modified in a controlled and predictable manner**. If two or more threads use it simultaneously, it will function correctly without needing additional synchronization from the user.

| **Technique** | **Thread Safety** | **Use Case** |
| --- | --- | --- |
| synchronized | ✅ Yes | Simple critical section control |
| java.util.concurrent | ✅ Yes | Collections and utilities |
| Atomic variables | ✅ Yes | Lock-free counters, flags |
| Immutable objects | ✅ Yes | Data that never changes |
| Thread-local storage | ✅ Yes | Per-thread variable isolation |
| Locks (ReentrantLock) | ✅ Yes | Fine-grained locking, tryLock, fairness, etc. |

### Techniques to Make Code Thread-Safe

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

### ReentrantLock implements Lock

* It is the implementation class of Lock interface and direct child class of object.
* Reentrant means a thread can acquire same lock multiple times without any issue.

### Comparable vs comparator

### 1.17.1 Comparable Interface:

The Comparable interface is used to define a *natural ordering* of objects. It’s implemented within the class itself, so objects of that class can be compared directly.

**int compareTo(T o);**

* This method compares the current object with the specified object o of the same type.
* The result of the comparison is an integer:
* A negative number if the current object is less than o
* Zero if they are equal
* A positive number if the current object is greater than o

### 1.17.2 Comparator Interface:

The Comparator interface is used to define a custom order or multiple possible orderings of objects. It is used when you need to compare objects in a different way than the natural ordering defined in the class or when the class does not implement Comparable.

**Method**: It has several methods, but the most important one is:

**int compare(T o1, T o2);**

This method compares two objects o1 and o2.

* Like compareTo, the result of the comparison is an integer:
  + A negative number if o1 is less than o2
  + Zero if they are equal
  + A positive number if o1 is greater than o2
* **Usage**: You use a Comparator when you want to sort by a different criterion or when you cannot modify the class to implement Comparable. Java 8 also introduced the ability to chain comparators and provide multiple ways to compare objects (e.g., using Comparator.comparing()).

### Java 8 Enhancements

Comparator in Java 8 is more powerful. You can use static methods like:

* Comparator.comparing()
* Comparator.thenComparing()
* Comparator.reversed()

people.sort(Comparator.comparing(Person::getAge).thenComparing(Person::getName));

people.forEach(person -> System.out.println(person.getName()));

A screenshot of a chat

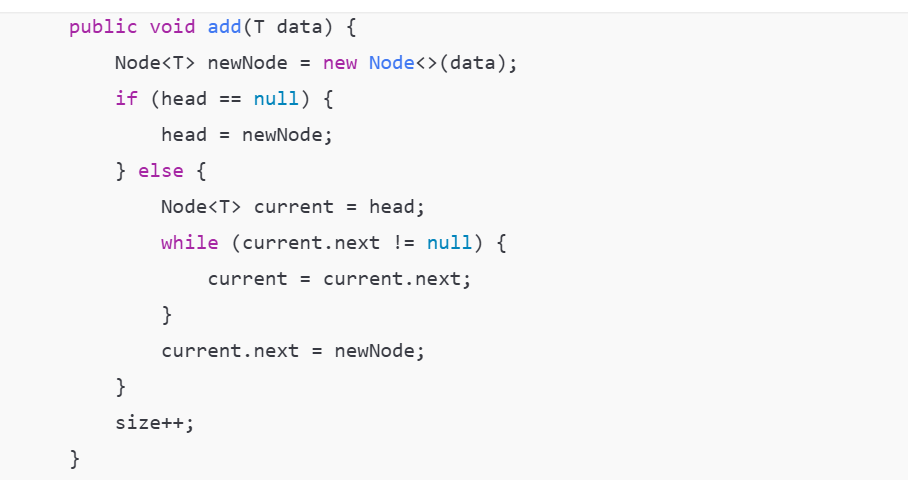
AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### Custom LinkedList Implementation





A computer code with text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect. A computer code with text

AI-generated content may be incorrect.

### What is callable and runnable interfaces

In Java, both Callable and Runnable are interfaces used to define tasks that can be executed by multiple threads, but they have some important differences.

**Runnable Interface**

Represents a task that **does not return a result** and **cannot throw a checked exception**.

public void run();

**Callable<V> Interface**

Introduced in **Java 5** as part of java.util.concurrent.

Represents a task that **returns a result** and **can throw a checked exception**.

It’s a **generic interface**, so you can specify the return type

Contains **one method**:

V call() throws Exception;

Usage with ExecutorService:

Callable<String> task = () -> {

return "Task result";

};

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String> future = executor.submit(task);

String result = future.get(); // blocks until the result is available

System.out.println(result);

A screenshot of a white screen

AI-generated content may be incorrect.

### Key Characteristics of Static Methods in Interfaces

* **Belongs to the Interface**: Static methods in interfaces belong to the interface itself, not the implementing class. They are not inherited by classes that implement the interface.
* **Cannot Be Overridden**: Since the method is static, it cannot be overridden by any class that implements the interface.
* **Can Be Called Directly on the Interface**: You call static methods directly on the interface, not on the instances of the implementing class.

### Key Characteristics of Default Methods

* **Implemented in the Interface**: Default methods have a method body, meaning you can provide a default implementation directly inside the interface.
* **Inheritable**: Default methods are inherited by implementing classes. If a class implements an interface with a default method, the class automatically gets the default implementation (unless the class overrides it).
* **Can Be Overridden**: If an implementing class does not want to use the default implementation, it can override the method like any other regular method.
* **No Need to Modify Implementing Classes**: The introduction of default methods was mainly designed to allow interfaces to evolve (add new methods) without breaking the classes that implement the interface. This is especially useful for APIs like Java's **Collection framework**.

### Method References

A **method reference** is essentially a shorthand notation for calling a method using a lambda expression.

Method references can be used in the context of **functional interfaces** (interfaces with a single abstract method), commonly used in places where **lambda expressions** are applicable.

Types of Method References

**Reference to a Static Method**

A static method in a class can be referred to with ClassName::staticMethodName.

**Reference to an Instance Method of a Particular Object**

You can refer to an instance method of an object using objectInstance::instanceMethodName.

**Reference to an Instance Method of an Arbitrary Object of a Particular Type**

This is useful when you don't have a specific object but want to refer to an instance method of an object of a specific type. It’s often used with **Streams**.

**Reference to a Constructor**

You can refer to a constructor using the ClassName::new syntax. This is particularly useful when using factory methods or Streams.

A screenshot of a computer

AI-generated content may be incorrect.

Method references provide a more concise and expressive way to refer to methods and are especially useful with Java's functional programming features (like **Streams**). They can simplify your code by replacing lambda expressions where the lambda body is just calling a method.

### Java11 features

* + 1. **var in Lambda Parameters**

You can now use var in lambda expressions

(var x, var y) -> x + y

* + 1. **HTTP Client API (Standardized)**

Introduced in Java 9 as incubating, now fully standardized.

Easier and more modern alternative to HttpURLConnection

* + 1. **Flight Recorder**

Lightweight performance and diagnostic tool, originally part of Oracle JDK, now open-sourced.

* + 1. **Z Garbage Collector (ZGC - Experimental)**

A scalable low-latency garbage collector (still experimental in Java 11).

* + 1. **New string methods**

isBlank() Checks if the string is empty or contains only whitespace characters.

lines() Splits the string into a **stream of lines** separated by line terminators (\n, \r, etc.).

strip() Similar to trim(), but more accurate with Unicode characters.

stripLeading() Removes **only leading** whitespaces.

stripTrailing() Removes **only trailing** whitespaces.

repeat(int count) Returns a string whose value is the original string repeated count times

* + 1. **Nest-Based Access Control**

Simplifies access between nested classes and their enclosing classes by treating them as a **nest**.

* + 1. **Dynamic Class-File Constants**

Supports a new constant pool form (CONSTANT\_Dynamic) to reduce boilerplate for dynamic languages

Removed Java EE and CORBA Modules

Removed javah javah tool was removed. Its functionality is now in javac

### Java21 features

### Virtual Thread

In Java, **Virtual Threads** are a new type of thread introduced as part of **Project Loom** to simplify concurrent programming by making it more scalable and lightweight.

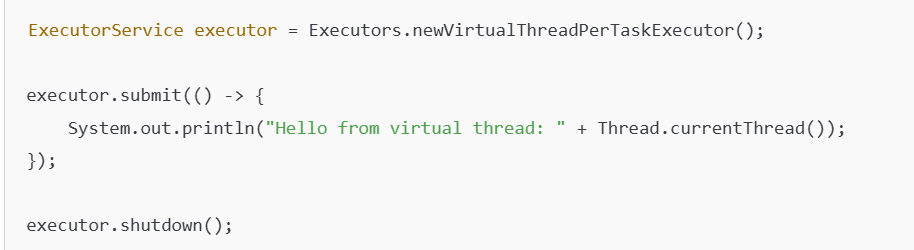
A **Virtual Thread** is a lightweight thread that is **managed by the Java Virtual Machine (JVM)** rather than the operating system (OS). Unlike traditional **platform threads** (which are mapped 1:1 to OS threads), virtual threads allow Java to run **millions of threads concurrently** with minimal resource usage.

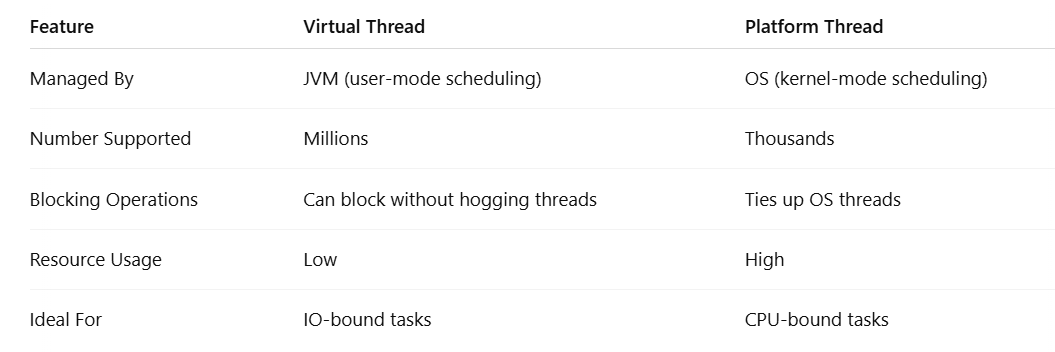
### Key Features

* **Lightweight**: Can spawn thousands or even millions of virtual threads.
* **Efficient Concurrency**: Ideal for high-concurrency applications like servers and web apps.
* **Non-blocking I/O**: Works well with structured concurrency and reactive programming.
* **Familiar API**: Uses the same Runnable, Callable, and Thread APIs.

A screen shot of a computer code

AI-generated content may be incorrect.





### What is Cohesion and Coupling

**Cohesion** is about **how strongly the components of a single module/class** are related to each other.

**Coupling** is about **how dependent one module/class is on another**.

| **Aspect** | **Cohesion** | **Coupling** |
| --- | --- | --- |
| **Definition** | Degree to which elements inside a module belong together | Degree of interdependence between modules |
| **Goal** | **High cohesion** is desirable | **Low coupling** is desirable |
| **Focus** | Within a class/module | Between classes/modules |
| **Improves** | Maintainability, readability, testability | Flexibility, reusability, modularity |
| **Good Example** | A class that performs a single, well-defined task | A class that depends on interfaces, not implementations |
| **Bad Example** | A class that handles DB, UI, and business logic | A class that directly instantiates and uses other classes |

### JVM\_OPTS

| **Option** | | | **Description** | **Example** | | |
| --- | --- | --- | --- | --- | --- | --- |
| -Xms<size> | | | Initial heap size | -Xms512m | | |
| -Xmx<size> | | | Maximum heap size | -Xmx2g | | |
| -Xss<size> | | | Stack size per thread | -Xss1m | | |
| -XX:MaxPermSize=<size> | | | Max size of Permanent Generation (Java 7 and below) | -XX:MaxPermSize=256m | | |
| -XX:MetaspaceSize=<size> | | | Initial metaspace size (Java 8+) | -XX:MetaspaceSize=128m | | |
| -XX:MaxMetaspaceSize=<size> | | | Maximum metaspace size (Java 8+) | -XX:MaxMetaspaceSize=512m | | |
| Option | | | Description | Example | | |
| -XX:+UseG1GC | | | Use G1 Garbage Collector |  | | |
| -XX:+UseParallelGC | | | Use Parallel GC |  | | |
| -XX:+UseConcMarkSweepGC | | | Use CMS Garbage Collector | (deprecated in Java 9+) | | |
| -XX:+PrintGCDetails | | | Print detailed GC logs |  | | |
| -Xlog:gc\* | | | GC logging (Java 9+) | -Xlog:gc\*:file=gc.log | | |
| -XX:+HeapDumpOnOutOfMemoryError | | | Dump heap on OOM |  | | |
| -XX:HeapDumpPath=<file> | | | File path for heap dump | -XX:HeapDumpPath=/tmp/heap.hprof | | |
| Option | | | Description | Example | | |
| -XX:+UseStringDeduplication | | | Enable string deduplication (G1GC only) |  | | |
| -XX:+TieredCompilation | | | Enable tiered compilation |  | | |
| -XX:+AlwaysPreTouch | | | Pre-touch memory pages (helps large heaps) |  | | |
| Option | | | Description | Example | | |
| -Xdebug | | | Enable debugging |  | | |
| -Xrunjdwp / -agentlib:jdwp | | | Java Debug Wire Protocol settings | -agentlib:jdwp=transport=dt\_socket,server=y,suspend=n,address=5005 | | |
| -verbose:gc | | | Print GC info to console |  | | |
| -Dcom.sun.management.jmxremote | | | Enable JMX remote monitoring |  | | |
| -Dcom.sun.management.jmxremote.port=<port> | | | JMX remote port | -Dcom.sun.management.jmxremote.port=9010 | | |
| Option | | | Description | Example | | |
| -D<name>=<value> | | | Set system property | -Denv=prod | | |
| -Duser.timezone=UTC | | | Set timezone | -Duser.timezone=UTC | | |
| -Dfile.encoding=UTF-8 | | | Set default file encoding | -Dfile.encoding=UTF-8 | | |
| Option | | | Description | Example | | |
| -cp or -classpath | | | Set classpath | -cp app.jar:lib/\* | | |
| -XX:+TraceClassLoading | | | Trace class loading |  | | |
| -XX:+PrintCompilation | | | Print JIT compilation activity |  | | |
| -Xnoclassgc | | | Disable class garbage collection |  | | |
| Option | | | Description | Example | | |
| -server / -client | | |  | Choose JVM mode -server | | |
| -jar | | | Run JAR file | -jar app.jar | | |
| --enable-preview | | |  | Enable Java preview features | | |
| --add-opens / --add-exports | | |  | Open modules (Java 9+) | | |
| **Mode** | **Startup Time** | **Optimization Level** | | | **Use Case** |
| -client | Fast | Low | | | GUI/Desktop apps |
| -server | Slower | High | | | Servers, services, APIs |

### JAVA\_INTERVIEW\_QUESTIONS

### JVM

The **Java Virtual Machine (JVM)** memory model is a crucial concept for understanding how Java applications run. It manages memory in several distinct regions, each serving a specific purpose. Here's an overview of the key memory areas: **Heap**, **Stack**, and **Metaspace**.

+------------------------------+

| Method Area (Metaspace) |

+------------------------------+

| Heap |

+------------------------------+

| Stack |

+------------------------------+

| Program Counter (PC) |

+------------------------------+

| Native Method Stack |

+------------------------------+

### Heap (Runtime Data Area)

* **Purpose**: Stores **objects** and **class instances**.
* **Shared** among all threads.
* Managed by the **Garbage Collector (GC)**.

What goes into the Heap?

Objects (e.g., new instances)

Arrays

Class variables (if part of an object)

Garbage Collection:

* VM automatically frees up heap memory using **GC algorithms** (e.g., G1GC, CMS).
* Heap is divided into:
* **Young Generation** (Eden + Survivor spaces)
* **Old Generation** (Tenured space)

### Stack (Java Stack or Thread Stack)

* **Purpose**: Stores **method call frames**, local variables, and partial results.
* **Private** to each thread (each thread gets its own stack).
* Stack memory is **fast** and **LIFO** (Last In, First Out).

Stack Frame Contains:

* Local variables
* Operand stack (for intermediate results)
* Return address

StackOverflowError: Occurs if recursion is too deep or if too many method calls are pushed without return.

### Metaspace (Replaces PermGen)

**Purpose**: Stores **class metadata**, such as structure of classes, methods, and annotations.

Introduced in **Java 8**, replaces **PermGen**.

Grows dynamically **outside of heap**.

What goes into Metaspace?

* Class metadata (e.g., class name, method signatures)
* Reflection data

**Key Feature:**

* **Metaspace uses native memory**, so its size is only limited by the OS (unless explicitly set using -XX:MaxMetaspaceSize).

### JVM Memory Areas Recap:

| **Area** | **Description** |
| --- | --- |
| **Heap** | Stores objects and class instances. |
| **Stack** | Stores method frames, local variables. |
| **Metaspace** (Java 8+) | Stores class metadata, static methods, and constants. |
| **Method Area** (part of Metaspace) | Stores class-level data like static variables, method info, and the string constant pool. |

### JVM Memory Diagram (Java 8+)

+----------------------------------------------+

| JVM Memory |

+----------------------+-----------------------+

| Heap | Metaspace |

| (for objects) | (replaces PermGen) |

| | |

| +---------------+ | +-----------------+ |

| | String Objects| | | String Constant | |

| | (e.g., new | | | Pool | |

| | String("a")) | | | ("a", "b", etc) | |

| +---------------+ | +-----------------+ |

| | |

| | +-----------------+ |

| | | Static final | |

| | | constants | |

| | | (e.g. int MAX = 5| |

| | | if inlined) | |

| | +-----------------+ |

+----------------------+-----------------------+

### Garbage Collection algorithms in Java

| **GC Algorithm** | **Pause Time** | **Throughput** | **Heap Size Suitability** | **Multithreaded** | **Concurrency** |
| --- | --- | --- | --- | --- | --- |
| Serial | High | Low | Small | ❌ No | ❌ No |
| Parallel | Medium | High | Medium to Large | ✅ Yes | ❌ No |
| CMS | Low | Medium | Medium | ✅ Yes | ✅ Partial |
| G1 | Low-Medium | High | Medium to Large | ✅ Yes | ✅ Partial |
| ZGC | Very Low | High | Huge (>1TB) | ✅ Yes | ✅ Full |
| Shenandoah | Very Low | High | Large | ✅ Yes | ✅ Full |

JVM options for GC logging:

-Xlog:gc\*

-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

### How does synchronized, ReentrantLock, ExecutorService work?

| **Feature** | **synchronized** | **ReentrantLock** | **ExecutorService** |
| --- | --- | --- | --- |
| Locking Type | Intrinsic (object/class) | Explicit (manual control) | Task Execution & Thread Management |
| Unlock Requirement | Automatic | Manual (unlock()) | Not applicable |
| Fairness | No | Optional (new ReentrantLock(true)) | Depends on pool |
| Try Lock / Timeout | No | Yes (tryLock(), tryLock(timeout)) | Not applicable |
| Interruptible | No | Yes | Yes (task cancellation APIs) |
| Use Case | Simple locking | Complex locking scenarios | Async task execution |

### How is ConcurrentHashMap implemented?

ConcurrentHashMap is a thread-safe and highly concurrent implementation of a hash map in Java. It allows multiple threads to read and write without locking the entire map. Here’s a breakdown of its implementation and internal working

* ConcurrentHashMap is backed by an array of Node<K,V>[] table, similar to HashMap.
* Each element in the array (a **bucket**) can be a linked list or a **tree** (similar to HashMap).
* It uses **fine-grained locking** and **non-blocking algorithms** for better concurrency.
* **No Global Locking**

Unlike Hashtable, ConcurrentHashMap does **not** synchronize every method or block the whole map.

* **Synchronized on Buckets**
* Write operations (like put) **synchronize on individual buckets** (i.e., the head node of a bucket).
* This allows multiple threads to modify different buckets concurrently.

**Reads Are Mostly Lock-Free**

* Reads (get) are **lock-free** and use **volatile** and **final** fields to ensure visibility and safety.

static class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

volatile V val;

volatile Node<K,V> next;

} The val and next fields are **volatile** to ensure visibility across threads.

**Treeification**

* If a bucket becomes too long (≥ 8 nodes), it is converted into a **balanced tree** (TreeBin) for faster lookups (like in HashMap).
* This helps maintain O(log n) lookup time under high collisions.
* Uses **CAS (Compare-And-Swap)** for some updates (especially during initialization of table or resizing) using Unsafe class.
* Helps in achieving non-blocking behavior for some operations.

### What are the best practices for designing immutable classes?

| **Principle** | **Benefit** |
| --- | --- |
| Final class | Prevents subclass modification |
| Private final fields | Prevents field mutation |
| Constructor initialization | Full object initialization |
| No setters | Ensures immutability |
| Defensive copies | Avoid shared mutable state |
| Immutable data structures | Safer multithreaded use |

### Java Memory Model (JMM)

**GC Algorithms in Java**

| **Collector** | **Description** | **JVM Flag** |
| --- | --- | --- |
| Serial GC | Single-threaded, best for small heaps | -XX:+UseSerialGC |
| Parallel GC | Multi-threaded, good throughput | -XX:+UseParallelGC |
| CMS (deprecated) | Concurrent, low pause | -XX:+UseConcMarkSweepGC |
| G1 GC | Balanced, low pause for large heaps | -XX:+UseG1GC |
| ZGC / Shenandoah | Ultra-low pause, scalable | -XX:+UseZGC, -XX:+UseShenandoahGC |

Java Heap Memory Structure

+------------------------+

| Heap |

| +------------------+ |

| | Young Generation | |

| | - Eden | |

| | - Survivor S0/S1| |

| +------------------+ |

| +------------------+ |

| | Old Generation | |

| +------------------+ |

| +------------------+ |

| | Metaspace | |

| +------------------+ |

+------------------------+

**JMM and Concurrency**

* **Volatile**: Ensures visibility across threads (no local caching).
* **Synchronized**: Adds both mutual exclusion and memory visibility.
* **Happens-before** relationship: Guarantees ordering of operations between threads.

**Common GC Tuning Parameters**

| Option | Description |
| --- | --- |
| -Xms | Initial heap size |
| -Xmx | Max heap size |
| -Xmn | Size of Young Generation |
| -XX:NewRatio | Ratio between Young and Old Gen |
| -XX:SurvivorRatio | Eden:Survivor ratio |
| -XX:MaxGCPauseMillis | Target pause time (used in G1, ZGC) |
| -XX:+PrintGCDetails | Log GC activity |
| -XX:+UseStringDeduplication | G1 only: reduce memory from duplicate Strings |

### How would you implement a thread-safe LRU cache?

Implementing a **thread-safe LRU (Least Recently Used) cache** in Java can be done in several ways. Here's a breakdown of the most common and effective approach using:

Approach: Use LinkedHashMap with synchronization

* LinkedHashMap maintains insertion/access order.
* Override removeEldestEntry to implement LRU eviction.
* Add synchronization or use Collections.synchronizedMap() or ReentrantReadWriteLock for thread safety.

**Implementation Overview**

| **Aspect** | **Details** |
| --- | --- |
| **Purpose** | Thread-safe in-memory LRU (Least Recently Used) cache |
| **Key Data Structure** | LinkedHashMap with access-order |
| **Thread Safety** | Achieved using Collections.synchronizedMap + synchronized blocks |
| **Eviction Policy** | Override removeEldestEntry() |
| **Access Order** | Enabled via LinkedHashMap(capacity, 0.75f, true) |
| **Max Capacity** | Passed in constructor |
| **Concurrency Control** | Method-level synchronization |
| **Alternative Approach** | Use ConcurrentHashMap + Doubly Linked List + ReentrantLock |

### Streams vs. For-Loops in Java

| **Aspect** | **Streams** | **Traditional For-Loops** |
| --- | --- | --- |
| **Readability** | More concise, declarative, expressive (map, filter, collect) | Verbose but explicit and easy to debug |
| **Performance (sequential)** | Slightly slower due to internal object creation and functional overhead | Typically faster due to low-level control |
| **Performance (parallel)** | Can outperform loops for large data sets using .parallelStream() | Manual parallelization is complex and error-prone |
| **Memory Allocation** | More memory usage (intermediate objects, lambdas) | Minimal memory footprint |
| **Short-circuiting** | Efficient for operations like anyMatch(), findFirst() | Requires explicit break or condition |
| **Index-based access** | Not ideal for indexed collections (like List) | Works best with indexed data |
| **Lazy Evaluation** | Yes – computation happens only when needed (terminal op) | No – each loop iteration is evaluated immediately |
| **Debugging** | Harder to step through (chained calls, lambdas) | Easier to debug with traditional breakpoints |
| **Side Effects** | Discouraged in functional style; less predictable | Easier to handle, though prone to bugs |
| **Best Use Case** | Transforming/processing collections, filtering large data | Simple loops, performance-critical or indexed operations |