**Java Multithreading**

**1. Thread class vs Runnable interface**

* **Thread class**: A concrete class that you can extend to create a thread. When you extend Thread, you can't extend any other class due to Java's single inheritance limitation.
* **Runnable interface**: A functional interface that you implement to define the task a thread should execute. It allows you to separate the task from the thread execution mechanism and supports composition better than inheritance.

**2. sleep() vs wait()**

* **sleep()**: A static method of Thread class that causes the current thread to pause execution for a specified time. It doesn't release locks/monitors held by the thread.
* **wait()**: An instance method of Object class that causes the current thread to wait until another thread calls notify()/notifyAll() on the same object. It releases the lock on the object so other threads can acquire it.

**3. Calling start() twice on the same thread object**

If you call start() twice on the same Thread object, it will throw an IllegalThreadStateException. A Thread object can be started only once; its lifecycle cannot be restarted once completed.

**4. How join() works**

The join() method causes the current thread to pause execution until the thread it's called on completes. If thread A calls thread B's join() method, thread A will wait until thread B finishes execution before continuing. This is useful when one thread depends on the results of another thread.

**5. The volatile keyword**

The volatile keyword ensures that a variable is always read from and written to main memory, not from thread-local caches. It provides memory visibility guarantees - any write to a volatile variable is immediately visible to other threads. However, it doesn't provide atomicity for compound operations.

**6. Synchronized block vs synchronized method**

* **Synchronized method**: Locks the entire method, using the current object (this) as the lock for instance methods or the Class object for static methods.
* **Synchronized block**: Allows you to specify exactly which object to use as the lock and minimize the critical section, potentially improving performance by reducing lock contention.

**7. Deadlock**

A deadlock occurs when two or more threads are blocked forever, each waiting for resources held by the other threads.

Scenario: Thread A holds lock on resource X and waits for resource Y, while Thread B holds lock on resource Y and waits for resource X. Neither thread can proceed.

**8. ReentrantLock vs synchronized block**

* **ReentrantLock**: More flexible with features like timed lock attempts, interruptible lock attempts, fairness policies, and non-block-structured locking.
* **Synchronized block**: Simpler to use with automatic lock release, but less flexible and without the advanced features of ReentrantLock.

**9. ExecutorService vs creating new threads**

ExecutorService provides thread pooling, which:

* Reduces overhead of thread creation/destruction
* Manages the number of concurrent threads
* Provides task queuing when all threads are busy
* Offers structured task submission and result handling
* Supports shutdown management and lifecycle control

**10. Fork/Join framework**

The Fork/Join framework is designed for recursive divide-and-conquer parallel processing. It uses a work-stealing algorithm where idle threads steal tasks from busy threads' queues, efficiently balancing workloads. It's particularly useful for CPU-intensive tasks that can be broken down into smaller subtasks.

**Java Exceptions**

**1. Checked vs unchecked exceptions**

* **Checked exceptions**: Extend Exception but not RuntimeException. They must be declared in a method's throws clause or handled with try-catch. Represent recoverable conditions.
* **Unchecked exceptions**: Extend RuntimeException or Error. They don't need to be declared or explicitly caught. Represent programming errors or unrecoverable conditions.

**2. Why catching Exception directly is bad**

Catching Exception directly is considered bad practice because:

* It catches all exceptions, including unchecked exceptions that might indicate programming errors
* It makes debugging harder by potentially hiding important errors
* It can lead to inappropriate error handling for different exception types
* It violates the principle of handling only exceptions you can actually recover from

**3. Multiple catch blocks**

Yes, you can have multiple catch blocks for a single try block to handle different types of exceptions differently:

try {

FileReader file = new FileReader("file.txt");

int data = 100 / Integer.parseInt(file.read());

file.close();

} catch (FileNotFoundException e) {

System.out.println("File not found");

} catch (ArithmeticException e) {

System.out.println("Arithmetic error");

} catch (IOException e) {

System.out.println("IO error");

}

**4. Return statement in finally block**

If a finally block has a return statement, it will override any return statement in the try or catch blocks. The method will always return the value from the finally block, which can be confusing and lead to unexpected behavior.

**5. throw vs throws**

* **throw**: Used to explicitly throw an exception within a method (e.g., throw new NullPointerException()).
* **throws**: Used in a method signature to declare exceptions that might be thrown but not handled within the method, passing the responsibility to the caller.

**6. Exception chaining**

Exception chaining preserves the original exception when throwing a new one. Example:

try {

dbConnection.query();

} catch (SQLException e) {

throw new ServiceException("Database query failed", e); // e is the cause

}

The ServiceException wraps the original SQLException, retaining the full error context.

**7. Empty catch blocks**

Empty catch blocks (exception swallowing) are generally bad practice because:

* They hide errors that might indicate serious problems
* They make debugging difficult as there's no information about what went wrong
* They can create subtle, hard-to-find bugs where the program continues with invalid state

**8. Custom exception use case**

Custom exceptions are useful when:

* You need domain-specific exception types for business logic
* You want to add additional context/fields relevant to your application
* You need to distinguish your application's exceptions from standard Java exceptions
* You want to create a hierarchical exception structure for your application

**9. Try-with-resources**

Try-with-resources automatically closes resources that implement AutoCloseable:

try (FileInputStream fis = new FileInputStream("file.txt");

BufferedReader br = new BufferedReader(new InputStreamReader(fis))) {

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

} // Resources automatically closed, even if exceptions occur

**10. RuntimeException for application-level exceptions**

Using RuntimeExceptions for application-level exceptions:

* Makes code cleaner by removing explicit exception handling where it's not useful
* Removes the need for throws declarations which can clutter method signatures
* But can make error handling less explicit and predictable
* May lead to uncaught exceptions if developers are unaware of potential exceptions
* Should be documented well to ensure proper handling at appropriate levels

**Java Collections**

**1. HashSet vs TreeSet**

**HashSet:**

* Implements the Set interface using a HashMap internally
* O(1) time complexity for basic operations (add, remove, contains)
* Does not maintain any specific order of elements
* Allows one null element
* Uses hashCode() and equals() methods for element comparison

**TreeSet:**

* Implements NavigableSet interface using a TreeMap (Red-Black tree)
* O(log n) time complexity for basic operations
* Maintains elements in sorted order (natural ordering or via a Comparator)
* Does not allow null elements
* Uses compareTo() or Comparator.compare() for element comparison

**2. LinkedHashMap for Insertion Order**

LinkedHashMap is preferred when maintaining insertion order because:

* It extends HashMap but adds a doubly-linked list that runs through all entries
* This linked list maintains the order in which keys were inserted
* Unlike HashMap (unordered) or TreeMap (sorted order), LinkedHashMap preserves the exact sequence of insertions
* The iteration order is predictable, making it useful for implementing LRU caches or maintaining historical record of entries
* It provides nearly identical performance as HashMap with only a slight overhead for maintaining the linked list

**3. ConcurrentHashMap vs HashMap**

**ConcurrentHashMap:**

* Thread-safe without locking the entire map
* Uses segment-level locking (pre-Java 8) or node-level locking (Java 8+)
* Allows concurrent reads and a configurable number of concurrent writes
* Does not allow null keys or values
* Provides atomic compound operations (putIfAbsent, replace, etc.)
* Iteration doesn't throw ConcurrentModificationException

**HashMap:**

* Not thread-safe
* Better performance in single-threaded environments
* Allows one null key and multiple null values
* Must be externally synchronized for thread safety
* Iteration may throw ConcurrentModificationException if modified during iteration

**4. ArrayList Internal Working When Exceeding Capacity**

When an element is added to an ArrayList beyond its capacity:

1. The add() method checks if the internal array needs to grow
2. If the current size equals capacity, ensureCapacity is triggered
3. A new capacity is calculated (typically: newCapacity = oldCapacity + (oldCapacity >> 1), which is roughly 1.5x growth)
4. A new, larger array is created
5. The existing elements are copied to the new array using Arrays.copyOf()
6. The old array is abandoned for garbage collection
7. The reference to the internal array is updated to point to the new array
8. Finally, the new element is added to the resized array

**5. Fail-Fast Behavior in Java Collections**

Fail-fast behavior:

* A safety mechanism that detects concurrent modification during iteration
* Collections maintain an internal "modification count" that's checked during iteration
* If the modification count changes during iteration, a ConcurrentModificationException is thrown
* Prevents potential corruption or unpredictable behavior from mid-iteration modifications
* It's a best-effort mechanism, not a guaranteed behavior
* Helps detect programming errors where collections are modified while being iterated
* Is not present in concurrent collections like ConcurrentHashMap which are designed for modification during iteration

**6. Collections.synchronizedList() and CopyOnWriteArrayList in Multithreading**

**Collections.synchronizedList():**

* Creates a thread-safe wrapper around a list by synchronizing on a mutex object
* Every method is synchronized, allowing only one thread to access the list at a time
* Good for balanced read/write scenarios
* Requires external synchronization during iteration
* Lower memory overhead compared to CopyOnWriteArrayList

**CopyOnWriteArrayList:**

* Creates a fresh copy of the underlying array for every modification
* Reads don't require locking, providing high concurrency for read operations
* Ideal for read-heavy, write-rare scenarios
* Iterators are guaranteed not to throw ConcurrentModificationException
* Iterators operate on a snapshot of the list at the point of iterator creation
* Higher memory overhead due to array copying

**7. Purpose of LinkedList vs ArrayList**

**LinkedList** is useful when:

* You need frequent insertions/deletions, especially in the middle of the list
* You need constant-time O(1) add/remove operations with iterators or at list ends
* You're implementing a queue or deque (LinkedList implements both Queue and Deque)
* You don't need random access by index (where it performs poorly with O(n) time)
* You're not concerned about the extra memory overhead of node objects

**ArrayList** is better when:

* You need frequent random access by index (O(1))
* Your list is predominantly used for reading rather than modification
* Memory efficiency is important
* You're doing bulk operations at the end of the list

**8. PriorityQueue Element Priority**

PriorityQueue determines priority by:

* Using natural ordering (elements must implement Comparable interface)
* Or using a custom Comparator provided during queue construction
* It's implemented as a binary heap data structure
* The head of the queue is always the smallest element according to the ordering
* When elements are added, they "bubble up" to their correct position
* When the head is removed, the structure is reorganized to maintain the heap property
* Elements with equal priority are not guaranteed any specific order
* It does not permit null elements

**9. Deque vs Queue**

**Queue:**

* Linear collection that supports element insertion at the tail and removal from the head
* Follows FIFO (First-In-First-Out) principle
* Main operations: add()/offer() to add to the tail, remove()/poll() to remove from the head

**Deque (Double-Ended Queue):**

* Extends Queue to support element insertion and removal at both ends
* Can function as both FIFO queue and LIFO stack
* Additional operations: addFirst()/offerFirst(), addLast()/offerLast()
* Additional removal operations: removeFirst()/pollFirst(), removeLast()/pollLast()
* More flexible but slightly more complex API
* Implementations include ArrayDeque and LinkedList

**10. WeakHashMap Beneficial Scenario**

A WeakHashMap is beneficial in:

* **Memory-sensitive caching:** When you need to cache objects but want them automatically removed when no longer referenced elsewhere.
* // Example: A cache of expensive image processing results
* Map<Image, ProcessedResult> cache = new WeakHashMap<>();

When images are no longer referenced in your application, they'll be automatically removed from the cache during garbage collection.

* **Object metadata storage:** When you want to associate metadata with objects without preventing them from being garbage collected.
* **Observer/Listener management:** When maintaining references to listeners that should be automatically cleaned up when the listeners are no longer used elsewhere.
* **Resource management:** When tracking resources that should be released when their associated objects are no longer in use.

**Java Collections**

**1. ArrayList vs LinkedList Performance**

* **ArrayList**: O(1) for random access, O(n) for insertions/deletions in middle, O(1) for end operations (amortized).
* **LinkedList**: O(n) for random access, O(1) for insertions/deletions with positioned iterators, O(1) for operations at both ends.
* **Key difference**: ArrayList excels at random access while LinkedList excels at insertions/deletions.

**2. HashMap Collision Handling**

* Uses chaining where colliding entries form a linked list in the same bucket.
* In Java 8+, converts linked lists to balanced trees when bucket size exceeds threshold (8).
* Improves worst-case performance from O(n) to O(log n) for large collision chains.

**3. TreeSet vs HashSet Usage**

* Use TreeSet when ordered iteration, range queries, or custom sorting is required.
* TreeSet maintains elements in sorted order using Red-Black tree implementation.
* TreeSet operations are O(log n) while HashSet are O(1), making HashSet faster for basic operations.

**4. LinkedHashMap Order Maintenance**

* Uses a doubly-linked list running through entries to maintain insertion order.
* Each entry contains next/previous pointers in addition to HashMap functionality.
* Can be configured to maintain access-order instead (useful for LRU caches).

**5. Comparator vs Comparable**

* **Comparable**: Internal comparison capability for natural ordering (compareTo method).
* **Comparator**: External comparison strategy allowing multiple orderings (compare method).
* Comparable is implemented by the class itself; Comparator is implemented separately.

**6. Vector vs ArrayList**

* Vector is synchronized (thread-safe) but slower; ArrayList is not synchronized but faster.
* Vector typically grows by doubling; ArrayList grows by 50%.
* Both are backed by resizable arrays with similar functionality.

**7. Collections.unmodifiableList()**

* Creates a read-only wrapper that throws exceptions for any modifying operations.
* Changes to the original list are still visible through the unmodifiable view.
* Used for defensive programming when exposing internal collections in APIs.

**8. PriorityQueue vs LinkedList**

* Use PriorityQueue when elements need to be processed by priority rather than FIFO order.
* PriorityQueue maintains a heap structure ensuring the highest/lowest priority element is always accessible.
* Useful for schedulers, resource allocation, and algorithms requiring priority-based processing.

**9. HashMap Resizing**

* Triggered when size exceeds capacity × load factor (default 0.75).
* Creates new array (typically double size) and rehashes all entries.
* In Java 8+, uses optimization where entries either stay at same position or move forward by oldCapacity.

**10. ConcurrentSkipListSet Usage**

* Provides thread-safe, lock-free access to a sorted set.
* Operations have expected O(log n) time complexity even with concurrent modification.
* Doesn't throw ConcurrentModificationException during iteration.
* Ideal when you need both concurrent access and sorted iteration.

**Searching in Java Collections**

**1. Efficient search in HashSet**

* Uses element's hashCode() to locate the bucket in O(1) time
* Then uses equals() to find exact match within bucket
* Provides constant-time performance regardless of size

**2. Arrays.binarySearch() algorithm and limitations**

* Uses binary search with O(log n) time complexity
* Limitations: array must be sorted, elements must be comparable, undefined behavior for unsorted arrays

**3. Collections.binarySearch() requiring sorted collections**

* Binary search algorithm requires order to eliminate half the elements each step
* With unsorted collections, algorithm makes wrong decisions about which portion to search
* Results are undefined and may miss target element if not sorted

**4. Finding index in ArrayList without built-in functions**

* Implement linear search with for-loop checking each element
* Compare using equals() method (handling null values properly)
* O(n) time complexity as each element must be checked sequentially

**5. TreeMap preferred over HashMap for searching**

* When range queries are needed (subMap, headMap, tailMap)
* When finding closest matches (ceilingKey, floorKey, higherKey, lowerKey)
* When maintaining sorted order is important for iteration

**6. LinkedList vs ArrayList searching**

* LinkedList requires sequential traversal from beginning/end (O(n))
* ArrayList provides constant-time random access via index (O(1))
* LinkedList lacks memory locality causing poor cache performance

**7. NavigableSet efficient searching**

* Provides specialized navigation methods (lower, floor, ceiling, higher)
* Supports range views and queries (headSet, tailSet, subSet)
* Offers bi-directional iteration with first/last element access

**8. TreeSet searching implementation**

* Uses balanced Red-Black tree with O(log n) search complexity
* Traverses tree from root, comparing target with nodes
* Goes left if target is smaller, right if larger, stops if equal

**9. ConcurrentSkipListMap advantages**

* Lock-free concurrent access with O(log n) expected search time
* No blocking when multiple threads search simultaneously
* Maintains sorted order while allowing concurrent modifications

**10. Reverse search using Collections.reverseOrder()**

* Sort collection using Collections.reverseOrder() comparator
* Use Collections.binarySearch with same reverse comparator
* Ensures consistent comparison criteria for both sorting and searching

**Sorting in Java Collections**

**1. Collections.sort() vs Arrays.sort()**

* Collections.sort() works on List implementations while Arrays.sort() works on arrays
* Collections.sort() uses merge sort (TimSort) for stability, Arrays.sort() uses quicksort for primitives and TimSort for objects
* Collections.sort() calls Arrays.sort() internally after converting the List to an array

**2. Natural ordering and Comparable**

* Natural ordering is the inherent ordering defined by a class itself
* Classes implement Comparable interface and override compareTo() method
* The compareTo() method returns negative, zero, or positive values to indicate less than, equal to, or greater than
* Used by sorting methods when no explicit Comparator is provided

**3. Comparator over Comparable use case**

* When sorting objects that don't implement Comparable
* When needing multiple different sort orders for the same class
* When sorting classes you can't modify (third-party libraries)
* When the natural ordering is different from what's needed for a specific use case

**4. TreeSet for sorted unique elements**

* Maintains elements in sorted order automatically (no explicit sort calls needed)
* Guarantees O(log n) time for add, remove, and contains operations
* Provides navigational methods like first(), last(), higher(), lower()
* Supports range-view operations (headSet, tailSet, subSet)
* Prevents duplicate elements like other Set implementations

**5. Sorting algorithms in Arrays.sort()**

* For primitive types: Dual-Pivot Quicksort (since Java 7)
* For object references: TimSort, a hybrid of merge sort and insertion sort
* TimSort is stable, Quicksort is not (but faster for primitives)
* Both have O(n log n) average time complexity

**6. PriorityQueue ordering maintenance**

* Implemented as a binary heap (complete binary tree)
* Parent always smaller/greater than children (min/max heap)
* New elements "bubble up" to correct position when added
* When removing head, last element moves to root then "sinks down"
* Ordering defined by natural order or custom Comparator

**7. Sorting custom objects on multiple fields**

* Create a Comparator that compares primary field first
* If primary field comparison equals zero, compare secondary field
* Chain comparisons using Comparator.comparing().thenComparing() (Java 8+)
* Or implement compare() method with multiple field comparisons

**8. Collections.shuffle() effects**

* Randomizes element order destroying any previous sorting
* Uses default or provided Random object for pseudorandom shuffling
* Uniformly permutes all possible orderings with equal likelihood
* Often used before sorting to avoid worst-case performance with nearly-sorted data

**9. Collections.sort() time complexity**

* Best case: O(n) when already nearly sorted
* Average case: O(n log n)
* Worst case: O(n log n)
* Space complexity: O(n) due to merge sort's need for auxiliary space
* Always stable (preserves relative order of equal elements)

**10. Sorting Map by values**

* Convert Map entries to List using new ArrayList(map.entrySet())
* Sort List using Comparator.comparing(Map.Entry::getValue)
* For preserved ordering, copy to LinkedHashMap in sorted order
* Alternatively use stream API: map.entrySet().stream().sorted(Map.Entry.comparingByValue())

**Collections and Wrapper Classes**

**1**. **Why is ArrayList preferred over LinkedList for random access?**

ArrayList uses a dynamic array, allowing O(1) time complexity for random access via index. LinkedList uses a doubly-linked list, requiring O(n) time to traverse to the desired node, making ArrayList faster for random access.

**2. Difference between HashMap and TreeMap?**

- \*\*HashMap\*\*: Stores key-value pairs in a hash table, offers O(1) average-case time for get/put operations, does not maintain order.

- \*\*TreeMap\*\*: Stores key-value pairs in a red-black tree, maintains keys in sorted order, offers O(log n) time for get/put operations.

**3. \*\*Why are Integer objects immutable in Java?\*\***

Integer objects are immutable to ensure thread safety, consistent behavior in collections (e.g., as HashMap keys), and to support caching (e.g., Integer.valueOf() reuses objects for values -128 to 127). Once created, their value cannot change.

**4. \*\*Why is Collections.unmodifiableList() used? Give one use case.\*\***

It creates an unmodifiable view of a list, preventing modifications while allowing read access. \*\*Use case\*\*: Returning a list from a method to ensure the caller cannot alter the internal state of an object (e.g., a configuration list in a class).

**5. \*\*Role of Comparator and Comparable in sorting collections?\*\***

- \*\*Comparable\*\*: Defines a natural ordering for a class by implementing compareTo(), used by collections like TreeSet or Collections.sort().

- \*\*Comparator\*\*: Provides a custom ordering via compare(), allowing sorting without modifying the class or sorting in multiple ways. Used in Collections.sort() or TreeMap.

**6. \*\*Why does Double.NaN == Double.NaN return false?\*\***

According to IEEE 754, NaN (Not a Number) is not equal to itself to handle undefined mathematical results consistently. Java follows this standard, so Double.NaN == Double.NaN returns false. Use Double.isNaN() to check for NaN.

**7. \*\*Internal working of HashSet in Java?\*\***

HashSet internally uses a HashMap, where elements are stored as keys with a dummy value (Object). It ensures uniqueness by checking hashCode() and equals(). Add/remove operations are O(1) on average, but performance depends on hash function quality.

**8. \*\*Why override equals() and hashCode() for custom objects in HashMap?\*\***

HashMap uses hashCode() to determine the bucket and equals() to check for key equality. Overriding ensures correct behavior for custom objects; otherwise, different instances with the same logical value may be treated as distinct keys, causing duplicates or retrieval issues.

**9. \*\*How is a PriorityQueue different from a Queue?\*\***

- \*\*Queue\*\*: Follows FIFO (or other ordering like LIFO), processes elements in insertion order (e.g., LinkedList as Queue).

- \*\*PriorityQueue\*\*: Orders elements based on natural ordering or a Comparator, always dequeuing the highest-priority element, using a min-heap (O(log n) for add/remove).

**10. \*\*Difference between Optional and null in Java?\*\***

- \*\*null\*\*: Represents absence of a value, can cause NullPointerException if not checked.

- \*\*Optional\*\*: A container that explicitly represents presence or absence of a value, encourages safe handling (e.g., via orElse(), isPresent()), reducing null-related errors.

- \*\*Example\*\*: Optional.ofNullable(x).orElse(defaultValue) safely handles null, unlike direct null checks.

**Private Constructor and Factory Pattern**

**1. \*\*Why would a class have a private constructor? Provide an example use case.\*\***

A private constructor restricts instantiation from outside the class, ensuring controlled object creation or preventing instantiation entirely.

\*\*Use case\*\*: A `Configuration` class with a private constructor to enforce a single instance loaded from a file, preventing external creation of multiple conflicting configurations.

**2. \*\*How can a private constructor be accessed using a Factory Pattern?\*\***

A Factory Pattern uses a static method (in the same class or a separate factory class) to invoke the private constructor and return an instance.

**3. \*\*Explain the Singleton Design Pattern using a private constructor.\*\***

The Singleton Pattern ensures a class has only one instance, accessed globally. A private constructor prevents external instantiation, and a static method manages the single instance.

\*\*Example\*\*:

public class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

}

The private constructor ensures only `getInstance()` creates or returns the instance.

**4. \*\*What is the purpose of the Factory Method Pattern?\*\***

The Factory Method Pattern defines an interface or abstract method for creating objects, allowing subclasses to specify the instantiated class. It decouples object creation from usage, enhancing flexibility.

\*\*Example\*\*: A `DocumentFactory` with `createDocument()` overridden by `PDFDocumentFactory` to create PDF documents.

**5. \*\*Why is a Factory Pattern preferred over simple object creation using new?\*\***

- \*\*Abstraction\*\*: Hides complex creation logic, making client code cleaner.

- \*\*Extensibility\*\*: Enables easy addition of new types without changing client code.

- \*\*Control\*\*: Centralizes object creation, allowing initialization or pooling.

**MVC Design Pattern**

**1. \*\*Role of the Controller in MVC Architecture\*\***

The Controller acts as an intermediary between the Model and the View. It handles user input, processes requests, updates the Model with new data, and ensures the View reflects the Model’s state. It interprets user actions (e.g., clicks, form submissions) and coordinates the flow of data, maintaining the application’s logic without directly containing business logic or UI code.

**2. \*\*How does the Model interact with the View in MVC?\*\***

The Model and View do not interact directly in a typical MVC architecture to maintain loose coupling. The Model holds the data and business logic, notifying the View of changes via an observer pattern (e.g., through listeners or events). The View queries the Model for data to display, but the Controller manages the updates and synchronization between them.

**3. \*\*Why is loose coupling achieved using MVC?\*\***

MVC promotes loose coupling by separating concerns: the Model (data/logic), View (UI), and Controller (input handling) are independent components. Changes in one (e.g., updating the View’s design) do not directly affect the others, as they communicate through well-defined interfaces or events, improving maintainability and testability.

**4. \*\*One real-world example where MVC is useful\*\***

\*\*Web Application\*\*: A blog platform uses MVC where the Model manages blog posts (database), the View displays posts (HTML/CSS), and the Controller handles user actions like creating or editing posts. This separation allows developers to update the UI or database logic independently.

**5. \*\*How can user input be validated in an MVC-based application?\*\***

- \*\*Controller\*\*: Validates input format (e.g., required fields, data types) before passing it to the Model.

- \*\*Model\*\*: Enforces business rules (e.g., valid email format, age restrictions) and rejects invalid data.

- \*\*View\*\*: Provides immediate feedback (e.g., error messages) based on validation results from the Controller or Model.

**Iterator and Decorator Pattern**

 **Advantage of using an Iterator over a simple for-loop**  
An Iterator provides a standardized way to traverse collections without exposing their internal structure, enabling polymorphism across different collection types (e.g., ArrayList, HashSet). It supports safe removal of elements during iteration (via remove()) and is more flexible for custom iteration logic, unlike a for-loop, which is tightly coupled to index-based or specific collection access.

 **How the Decorator Pattern follows the Open/Closed Principle**  
The Decorator Pattern allows extending an object’s behavior without modifying its class, adhering to the Open/Closed Principle (open for extension, closed for modification). Decorators wrap the original object, adding new functionality via composition, so the base class remains unchanged while new behaviors are added through new decorator classes.

 **How does Iterator.remove() work?**  
The Iterator.remove() method safely removes the last element returned by Iterator.next() from the underlying collection during iteration. It must be called after next() and before the next next() call, ensuring the collection is modified without causing concurrency issues (e.g., ConcurrentModificationException). It’s supported only if the Iterator’s implementation allows modification.

**Difference between ListIterator and Iterator**

* **Iterator**: Provides forward-only traversal of any collection, with methods like next(), hasNext(), and optional remove().
* **ListIterator**: Extends Iterator for List implementations (e.g., ArrayList), adding backward traversal (previous(), hasPrevious()), modification (set(), add()), and index access (nextIndex(), previousIndex()). It’s specific to lists and more powerful for bidirectional navigation and editing.

**Strategy and Annotation**

 **How does the Strategy Pattern promote flexibility in code design?**

The Strategy Pattern promotes flexibility by encapsulating a family of interchangeable algorithms (strategies) in separate classes, allowing clients to switch strategies dynamically at runtime via composition. This decouples the algorithm implementation from the client, enabling easy addition or modification of strategies without altering the core logic.

 **What problem is solved by using the Strategy Pattern?**

The Strategy Pattern solves the problem of rigid, hard-coded behavior in a class that needs to support multiple algorithms or behaviors. It avoids code duplication and complex conditional logic (e.g., multiple if-else statements), making it easier to extend, maintain, and test different behaviors independently.

 **Explain the use of @Override annotation in Java.**

The @Override annotation indicates that a method is intended to override a method in a superclass or implement an interface method. It ensures compile-time checking to catch errors (e.g., mismatched method signatures), improving code reliability and readability by explicitly declaring the override intent.

 **How is @FunctionalInterface used in Java?**

The @FunctionalInterface annotation marks an interface as a functional interface, which must have exactly one abstract method (excluding default or static methods). It enables the interface to be used with lambda expressions or method references, ensuring clarity and preventing accidental addition of multiple abstract methods.

**Purpose of RetentionPolicy.RUNTIME in custom annotations**

* **Purpose**: Specifies that the custom annotation is retained at runtime, allowing it to be accessed via reflection.
* **Use**: Enables runtime processing, such as reading annotation metadata in frameworks (e.g., Spring, Hibernate).
* **Example**: @Retention(RetentionPolicy.RUNTIME) on a custom annotation like @MyAnnotation ensures it’s available for runtime inspection, unlike SOURCE or CLASS policies.

**Generics and UML Diagrams**

 **Concept of type erasure in Java Generics**  
Type erasure is the process by which Java’s compiler removes generic type information during compilation, replacing generic types with their raw types or bounds (e.g., List<T> becomes List). This ensures backward compatibility with pre-generics code but means generic type details are unavailable at runtime, relying on compile-time type checking for safety.

** \*\*Why is List<?> used in Java?\*\* `**

List<?>represents an unbounded wildcard, denoting a list of unknown type. It’s used when the specific type of elements is irrelevant, allowing flexibility in methods that operate on lists generically while ensuring type safety by restricting modifications (e.g., cannot add elements exceptnull`). It’s useful for read-only operations like printing a list.

 **Difference between List<? extends Number> and List<? super Number>?**

* **List<? extends Number>**: A list of any type that is a subtype of Number (e.g., List<Integer>, List<Double>). It allows reading elements as Number but prohibits adding elements (except null) due to type uncertainty.
* **List<? super Number>**: A list of any type that is a supertype of Number (e.g., List<Number>, List<Object>). It allows adding Number or its subtypes but restricts reading to Object due to unknown specific type.

**API Handling**

1. **Key methods of the HttpURLConnection class in Java**
   * setRequestMethod(String method): Sets the HTTP method (e.g., GET, POST).
   * setRequestProperty(String key, String value): Adds headers (e.g., Content-Type).
   * setDoOutput(boolean value): Enables output for POST/PUT requests.
   * getInputStream(): Retrieves the response input stream.
   * getResponseCode(): Returns the HTTP status code (e.g., 200).
   * disconnect(): Closes the connection.
2. **Handling RESTful API requests using Java**  
   RESTful API requests can be made using HttpURLConnection or libraries like OkHttp/RestTemplate. For HttpURLConnection:
   * Create a URL object for the endpoint.
   * Open an HttpURLConnection, set the request method (GET, POST, etc.), and headers.
   * For POST/PUT, write data to the output stream.
   * Read the response using getInputStream() or getErrorStream().  
     **Example**: Use HttpClient (Java 11+) for a cleaner approach with asynchronous support.
3. **Role of BufferedReader in API response reading**  
   BufferedReader reads the API response stream efficiently by buffering input, reducing direct I/O operations. It reads text line-by-line from the InputStreamReader (wrapping HttpURLConnection.getInputStream()), making it easier to process large responses or JSON/XML data.
4. **Parsing JSON responses using Java**  
   JSON responses can be parsed using libraries like Jackson or Gson. With Jackson:
   * Use ObjectMapper to deserialize JSON into Java objects or read it as a JsonNode.
5. **Exceptions to handle when making HTTP requests in Java**
   * IOException: For network failures or stream issues.
   * MalformedURLException: For invalid URLs.
   * ProtocolException: For invalid HTTP methods.
   * SecurityException: For restricted connections.
   * JsonProcessingException (with JSON parsing): For invalid JSON data.
6. **Measuring API response time in Java**  
   Measure response time by recording timestamps before and after the request:

*long startTime = System.nanoTime();*

*// Make API request*

*HttpURLConnection conn = (HttpURLConnection) new URL("https://api.example.com").openConnection();*

*conn.getResponseCode();*

*long endTime = System.nanoTime();*

*long durationMs = (endTime - startTime) / 1\_000\_000;*

1. **Concept of connection pooling in API handling**  
   Connection pooling reuses existing HTTP connections to reduce overhead from establishing new connections (e.g., TCP handshakes). Libraries like Apache HttpClient or OkHttp manage a pool of connections, improving performance for frequent requests. Java’s HttpClient (Java 11+) supports pooling implicitly, maintaining connections based on keep-alive settings.
2. **Significance of status codes like 200, 404, and 500**
   * **200 OK**: Request succeeded, response contains data.
   * **404 Not Found**: Requested resource doesn’t exist on the server.
   * **500 Internal Server Error**: Server encountered an unexpected error, indicating a server-side issue.
3. **Handling authentication in API requests using Java**  
   Authentication can be implemented by:
   * **Basic Auth**: Encode username:password in Base64 and set the Authorization header ("Basic " + encodedCredentials).
   * **Bearer Token**: Add a token in the Authorization header ("Bearer " + token).
   * **API Key**: Include the key in headers or query parameters.
4. **Implementing API rate limiting in Java**

* Use a library like Bucket4j or implement a token bucket algorithm.
* Track request counts/timestamps per client (e.g., using a Map or database).
* Before each request, check if the limit (e.g., 100 requests/hour) is exceeded; delay or reject if necessary.  
  **Example**: With Bucket4j, configure a bucket with a rate (e.g., 10 requests/minute) and consume tokens before sending requests, blocking if no tokens are available.