

# **Atlassian Java Coding & Design Round Interview Preparation**

## **Problem 1: Employee Directory - Closest Common Group**

### **Problem Analysis**

Find the closest common parent group for a set of employees in an organizational hierarchy using Java.

### **Solution Approach**

Use tree data structure with LCA (Lowest Common Ancestor) algorithm implemented in Java.

java

```
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import java.util.stream.Collectors;

class Group {
    private String name;
    private Group parent;
    private Set<Group> children;
    private Set<String> employees;

    public Group(String name) {
        this.name = name;
        this.children = ConcurrentHashMap.newKeySet();
        this.employees = ConcurrentHashMap.newKeySet();
    }

    // Getters and setters
    public String getName() { return name; }
    public Group getParent() { return parent; }
    public void setParent(Group parent) { this.parent = parent; }
    public Set<Group> getChildren() { return children; }
    public Set<String> getEmployees() { return employees; }

    public void addChild(Group child) {
        children.add(child);
        child.setParent(this);
    }

    public void addEmployee(String employee) {
        employees.add(employee);
    }
}

class Employee {
    private String name;
    private Set<Group> groups;

    public Employee(String name, Group group) {
        this.name = name;
        this.groups = ConcurrentHashMap.newKeySet();
        this.groups.add(group);
        group.addEmployee(name);
    }

    public String getName() { return name; }
    public Set<Group> getGroups() { return groups; }
```

```

    public void addToGroup(Group group) {
        groups.add(group);
        group.addEmployee(name);
    }
}

```

```

public class EmployeeDirectory {
    private final Map<String, Group> groups;
    private final Map<String, Employee> employees;
    private final ReentrantReadWriteLock lock;

```

```

    public EmployeeDirectory() {
        this.groups = new ConcurrentHashMap<>();
        this.employees = new ConcurrentHashMap<>();
        this.lock = new ReentrantReadWriteLock();
    }

```

```

    public void addGroup(String name, String parentName) {
        lock.writeLock().lock();
        try {
            Group group = new Group(name);
            groups.put(name, group);

            if (parentName != null && groups.containsKey(parentName)) {
                Group parent = groups.get(parentName);
                parent.addChild(group);
            }
        } finally {
            lock.writeLock().unlock();
        }
    }

```

```

    public void addEmployee(String empName, String groupName) {
        lock.writeLock().lock();
        try {
            if (!groups.containsKey(groupName)) {
                return;
            }

            Group group = groups.get(groupName);
            if (employees.containsKey(empName)) {
                employees.get(empName).addToGroup(group);
            } else {
                employees.put(empName, new Employee(empName, group));
            }
        } finally {
            lock.writeLock().unlock();
        }
    }
}

```

```

private List<Group> getPathToRoot(Group group) {
    List<Group> path = new ArrayList<>();
    Group current = group;

    while (current != null) {
        path.add(current);
        current = current.getParent();
    }

    return path;
}

private Group findLCA(List<Group> groups) {
    if (groups.isEmpty()) {
        return null;
    }

    // Get all paths to root
    List<List<Group>> paths = groups.stream()
        .map(this::getPathToRoot)
        .collect(Collectors.toList());

    int minLength = paths.stream()
        .mapToInt(List::size)
        .min()
        .orElse(0);

    // Find LCA by comparing from root down
    for (int i = 0; i < minLength; i++) {
        Group currentGroup = paths.get(0).get(paths.get(0).size() - 1 - i);
        boolean allMatch = paths.stream()
            .allMatch(path -> path.get(path.size() - 1 - i).equals(currentGroup));

        if (!allMatch) {
            return i > 0 ? paths.get(0).get(paths.get(0).size() - i) : null;
        }
    }

    return minLength > 0 ? paths.get(0).get(paths.get(0).size() - minLength) : null;
}

public String getCommonGroupForEmployees(List<String> employeeNames) {
    lock.readLock().lock();
    try {
        Set<Group> allGroups = new HashSet<>();

        // Collect all groups for all employees
        for (String empName : employeeNames) {

```

```

        if (employees.containsKey(empName)) {
            allGroups.addAll(employees.get(empName).getGroups());
        }
    }

    if (allGroups.isEmpty()) {
        return null;
    }

    Group lca = findLCA(new ArrayList<>(allGroups));
    return lca != null ? lca.getName() : null;
} finally {
    lock.readLock().unlock();
}
}

// Usage example
public static void main(String[] args) {
    EmployeeDirectory directory = new EmployeeDirectory();
    directory.addGroup("Engineering", null);
    directory.addGroup("Backend", "Engineering");
    directory.addGroup("Frontend", "Engineering");
    directory.addEmployee("Alice", "Backend");
    directory.addEmployee("Bob", "Frontend");

    String result = directory.getCommonGroupForEmployees(
        Arrays.asList("Alice", "Bob")
    );
    System.out.println("Common group: " + result); // Output: Engineering
}
}

```

## Problem 2: Tennis Court Booking System

### Java Implementation with Priority Queues and Maintenance Logic

java

```
import java.util.*;
import java.util.stream.Collectors;

class BookingRecord {
    private int id;
    private int startTime;
    private int finishTime;

    public BookingRecord(int id, int startTime, int finishTime) {
        this.id = id;
        this.startTime = startTime;
        this.finishTime = finishTime;
    }

    // Getters
    public int getId() { return id; }
    public int getStartTime() { return startTime; }
    public int getFinishTime() { return finishTime; }

    @Override
    public String toString() {
        return String.format("Booking{id=%d, start=%d, end=%d}", id, startTime, finishTime);
    }
}

class Court {
    private int id;
    private int availableTime;
    private int bookingsCount;
    private List<BookingRecord> bookings;

    public Court(int id) {
        this.id = id;
        this.availableTime = 0;
        this.bookingsCount = 0;
        this.bookings = new ArrayList<>();
    }

    // Getters and setters
    public int getId() { return id; }
    public int getAvailableTime() { return availableTime; }
    public void setAvailableTime(int availableTime) { this.availableTime = availableTime; }
    public int getBookingsCount() { return bookingsCount; }
    public void setBookingsCount(int bookingsCount) { this.bookingsCount = bookingsCount; }
    public List<BookingRecord> getBookings() { return bookings; }

    public void addBooking(BookingRecord booking) {
```

```

        bookings.add(booking);
        bookingsCount++;
    }

    @Override
    public String toString() {
        return String.format("Court{id=%d, available=%d, bookings=%d}",
            id, availableTime, bookingsCount);
    }
}

public class TennisCourtScheduler {
    private List<Court> courts;
    private int courtCounter;

    public TennisCourtScheduler() {
        this.courts = new ArrayList<>();
        this.courtCounter = 0;
    }

    public List<Court> assignCourtsBasic(List<BookingRecord> bookingRecords) {
        // Sort bookings by start time
        List<BookingRecord> sortedBookings = bookingRecords.stream()
            .sorted(Comparator.comparingInt(BookingRecord::getStartTime))
            .collect(Collectors.toList());

        // Priority queue to track when courts become available
        PriorityQueue<Court> availableCourts = new PriorityQueue<>(
            Comparator.comparingInt(Court::getAvailableTime)
        );

        for (BookingRecord booking : sortedBookings) {
            Court court = null;

            // Check if any court is available
            if (!availableCourts.isEmpty() &&
                availableCourts.peek().getAvailableTime() <= booking.getStartTime()) {
                court = availableCourts.poll();
            } else {
                // Create new court
                court = new Court(courtCounter++);
                courts.add(court);
            }

            // Assign booking to court
            court.addBooking(booking);
            court.setAvailableTime(booking.getFinishTime());

            // Add court back to queue
            availableCourts.offer(court);
        }
    }
}

```

```

    }

    return courts;
}

public List<Court> assignCourtsWithMaintenance(
    List<BookingRecord> bookingRecords,
    int maintenanceTime,
    int durability) {

    List<BookingRecord> sortedBookings = bookingRecords.stream()
        .sorted(Comparator.comparingInt(BookingRecord::getStartTime))
        .collect(Collectors.toList());

    PriorityQueue<Court> availableCourts = new PriorityQueue<>(
        Comparator.comparingInt(Court::getAvailableTime)
    );

    for (BookingRecord booking : sortedBookings) {
        Court suitableCourt = null;
        List<Court> tempCourts = new ArrayList<>();

        // Find available court that doesn't need maintenance
        while (!availableCourts.isEmpty()) {
            Court court = availableCourts.poll();

            if (court.getAvailableTime() <= booking.getStartTime() &&
                court.getBookingsCount() < durability) {
                suitableCourt = court;
                break;
            } else {
                tempCourts.add(court);
            }
        }

        // Restore remaining courts to queue
        availableCourts.addAll(tempCourts);

        if (suitableCourt == null) {
            // Create new court
            suitableCourt = new Court(courtCounter++);
            courts.add(suitableCourt);
        }

        // Assign booking
        suitableCourt.addBooking(booking);

        // Calculate next available time
        int nextAvailable = booking.getFinishTime();
    }
}

```



```

        if (suitableCourt.getBookingsCount() >= durability) {
            nextAvailable += maintenanceTime;
            suitableCourt.setBookingsCount(0); // Reset after maintenance
        }

        suitableCourt.setAvailableTime(nextAvailable);
        availableCourts.offer(suitableCourt);
    }

    return courts;
}

public int minCourtsNeeded(List<BookingRecord> bookingRecords) {
    List<int[]> events = new ArrayList<>();

    // Create start and end events
    for (BookingRecord booking : bookingRecords) {
        events.add(new int[]{booking.getStartTime(), 1}); // Start event
        events.add(new int[]{booking.getFinishTime(), -1}); // End event
    }

    events.sort((a, b) -> {
        if (a[0] == b[0]) {
            return Integer.compare(a[1], b[1]); // End events before start events
        }
        return Integer.compare(a[0], b[0]);
    });

    int currentCourts = 0;
    int maxCourts = 0;

    for (int[] event : events) {
        currentCourts += event[1];
        maxCourts = Math.max(maxCourts, currentCourts);
    }

    return maxCourts;
}

public boolean checkBookingConflict(BookingRecord booking1, BookingRecord booking2) {
    return !(booking1.getFinishTime() <= booking2.getStartTime() ||
        booking2.getFinishTime() <= booking1.getStartTime());
}

// Usage example
public static void main(String[] args) {
    TennisCourtScheduler scheduler = new TennisCourtScheduler();
    List<BookingRecord> bookings = Arrays.asList(
        new BookingRecord(1, 9, 12),
        new BookingRecord(2, 10, 13)
    );

```

```
        new BookingRecord(2, 10, 13),
        new BookingRecord(3, 11, 14),
        new BookingRecord(4, 15, 17)
    );

    List<Court> courts = scheduler.assignCourtsBasic(bookings);
    System.out.println("Number of courts needed: " + courts.size());

    courts.forEach(court -> {
        System.out.println(court + " -> " + court.getBookings());
    });

    System.out.println("Minimum courts needed: " + scheduler.minCourtsNeeded(bookings));
}
}
```

## Problem 3: Commodity Price Tracking

### Java Implementation with Thread-Safe Operations

java

```
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantLock;

public class CommodityPriceTracker {
    private final Map<Integer, Double> prices;
    private final PriorityQueue<PriceEntry> maxHeap;
    private final Set<PriceEntry> deletedEntries;
    private final ReentrantLock lock;
    private double currentMaxPrice;
    private Integer maxTimestamp;

    private static class PriceEntry {
        final double price;
        final int timestamp;

        PriceEntry(double price, int timestamp) {
            this.price = price;
            this.timestamp = timestamp;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            PriceEntry that = (PriceEntry) obj;
            return Double.compare(that.price, price) == 0 && timestamp == that.timestamp;
        }

        @Override
        public int hashCode() {
            return Objects.hash(price, timestamp);
        }

        @Override
        public String toString() {
            return String.format("PriceEntry{price=%.2f, timestamp=%d}", price, timestamp);
        }
    }

    public CommodityPriceTracker() {
        this.prices = new ConcurrentHashMap<>();
        this.maxHeap = new PriorityQueue<>((a, b) -> Double.compare(b.price, a.price));
        this.deletedEntries = ConcurrentHashMap.newKeySet();
        this.lock = new ReentrantLock();
        this.currentMaxPrice = 0.0;
        this.maxTimestamp = null;
    }
}
```

```

    }

    public void updatePrice(int timestamp, double price) {
        lock.lock();
        try {
            Double oldPrice = prices.get(timestamp);
            prices.put(timestamp, price);

            // Add new entry to heap
            PriceEntry newEntry = new PriceEntry(price, timestamp);
            maxHeap.offer(newEntry);

            // If we had an old price, mark it as deleted
            if (oldPrice != null) {
                deletedEntries.add(new PriceEntry(oldPrice, timestamp));
            }

            // Update current max if necessary
            updateCurrentMax();
        } finally {
            lock.unlock();
        }
    }

    private void updateCurrentMax() {
        // Clean heap and update current max
        while (!maxHeap.isEmpty() && deletedEntries.contains(maxHeap.peek())) {
            PriceEntry removed = maxHeap.poll();
            deletedEntries.remove(removed);
        }

        if (!maxHeap.isEmpty()) {
            PriceEntry maxEntry = maxHeap.peek();
            currentMaxPrice = maxEntry.price;
            maxTimestamp = maxEntry.timestamp;
        } else {
            currentMaxPrice = 0.0;
            maxTimestamp = null;
        }
    }

    public double getMaxCommodityPrice() {
        lock.lock();
        try {
            updateCurrentMax();
            return currentMaxPrice;
        } finally {
            lock.unlock();
        }
    }
}

```

```

    public Optional<Double> getPriceAtTimestamp(int timestamp) {
        return Optional.ofNullable(prices.get(timestamp));
    }

    public List<Map.Entry<Integer, Double>> getPriceHistorySorted() {
        return prices.entrySet().stream()
            .sorted(Map.Entry.comparingByKey())
            .collect(ArrayList::new, (list, entry) -> list.add(entry), ArrayList::addAll);
    }

    public Map<String, Object> getStatistics() {
        lock.lock();
        try {
            updateCurrentMax();
            OptionalDouble average = prices.values().stream()
                .mapToDouble(Double::doubleValue)
                .average();

            Map<String, Object> stats = new HashMap<>();
            stats.put("totalEntries", prices.size());
            stats.put("maxPrice", currentMaxPrice);
            stats.put("maxTimestamp", maxTimestamp);
            stats.put("averagePrice", average.orElse(0.0));

            return stats;
        } finally {
            lock.unlock();
        }
    }
}

```

*// Usage example*

```

public static void main(String[] args) {
    CommodityPriceTracker tracker = new CommodityPriceTracker();

    tracker.updatePrice(100, 25.5);
    tracker.updatePrice(105, 30.0);
    tracker.updatePrice(102, 28.0);
    tracker.updatePrice(100, 26.0); // Update existing timestamp

    System.out.println("Max price: " + tracker.getMaxCommodityPrice()); // 30.0
    System.out.println("Statistics: " + tracker.getStatistics());

    System.out.println("Price history:");
    tracker.getPriceHistorySorted().forEach(entry ->
        System.out.println("Timestamp: " + entry.getKey() + ", Price: " + entry.getValue())
    );
}
}

```

---

## Problem 4: Popular Content Tracking

### Java Implementation with Efficient Max Tracking

java

```
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantLock;

public class PopularContentTracker {
    private final Map<Integer, Integer> popularity;
    private final PriorityQueue<ContentEntry> maxHeap;
    private final Set<ContentEntry> deletedEntries;
    private final ReentrantLock lock;
    private int currentMaxPopularity;
    private int currentMaxContentId;

    private static class ContentEntry {
        final int popularity;
        final int contentId;

        ContentEntry(int popularity, int contentId) {
            this.popularity = popularity;
            this.contentId = contentId;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            ContentEntry that = (ContentEntry) obj;
            return popularity == that.popularity && contentId == that.contentId;
        }

        @Override
        public int hashCode() {
            return Objects.hash(popularity, contentId);
        }

        @Override
        public String toString() {
            return String.format("ContentEntry{popularity=%d, contentId=%d}",
                popularity, contentId);
        }
    }

    public PopularContentTracker() {
        this.popularity = new ConcurrentHashMap<>();
        this.maxHeap = new PriorityQueue<>((a, b) -> Integer.compare(b.popularity, a.popularity));
        this.deletedEntries = ConcurrentHashMap.newKeySet();
        this.lock = new ReentrantLock();
        this.currentMaxPopularity = 0;
    }
}
```

```

    this.currentMaxContentId = -1;
}

public void increasePopularity(int contentId) {
    lock.lock();
    try {
        int oldPopularity = popularity.getOrDefault(contentId, 0);
        int newPopularity = oldPopularity + 1;
        popularity.put(contentId, newPopularity);

        // Mark old entry as deleted if it exists
        if (oldPopularity > 0) {
            deletedEntries.add(new ContentEntry(oldPopularity, contentId));
        }

        // Add new entry to heap
        ContentEntry newEntry = new ContentEntry(newPopularity, contentId);
        maxHeap.offer(newEntry);

        // Update current max if this is now the highest
        if (newPopularity > currentMaxPopularity) {
            currentMaxPopularity = newPopularity;
            currentMaxContentId = contentId;
        }
    } finally {
        lock.unlock();
    }
}

public void decreasePopularity(int contentId) {
    lock.lock();
    try {
        if (!popularity.containsKey(contentId) || popularity.get(contentId) <= 0) {
            return;
        }

        int oldPopularity = popularity.get(contentId);
        int newPopularity = oldPopularity - 1;

        // Mark old entry as deleted
        deletedEntries.add(new ContentEntry(oldPopularity, contentId));

        // Update popularity map
        if (newPopularity > 0) {
            popularity.put(contentId, newPopularity);
            maxHeap.offer(new ContentEntry(newPopularity, contentId));
        } else {
            popularity.put(contentId, 0);
        }
    }
}

```



```

        // If this was the max content, we need to recalculate
        if (contentId == currentMaxContentId && oldPopularity == currentMaxPopularity) {
            recalculateMax();
        }
    } finally {
        lock.unlock();
    }
}

```

```

private void recalculateMax() {
    // Clean up deleted entries from heap
    while (!maxHeap.isEmpty() && deletedEntries.contains(maxHeap.peek())) {
        ContentEntry removed = maxHeap.poll();
        deletedEntries.remove(removed);
    }

    if (!maxHeap.isEmpty()) {
        ContentEntry maxEntry = maxHeap.peek();
        currentMaxPopularity = maxEntry.popularity;
        currentMaxContentId = maxEntry.contentId;
    } else {
        // Find max from current popularity map
        int maxPop = 0;
        int maxContent = -1;

        for (Map.Entry<Integer, Integer> entry : popularity.entrySet()) {
            if (entry.getValue() > maxPop) {
                maxPop = entry.getValue();
                maxContent = entry.getKey();
            }
        }

        currentMaxPopularity = maxPop;
        currentMaxContentId = maxPop > 0 ? maxContent : -1;
    }
}

```

```

public int getMostPopularContent() {
    lock.lock();
    try {
        recalculateMax();
        return currentMaxPopularity > 0 ? currentMaxContentId : -1;
    } finally {
        lock.unlock();
    }
}

```

```

public int getPopularity(int contentId) {
    return popularity.getDefault(contentId, 0);
}

```

```

    }

    public List<Map.Entry<Integer, Integer>> getTopContent(int limit) {
        lock.lock();
        try {
            return popularity.entrySet().stream()
                .filter(entry -> entry.getValue() > 0)
                .sorted((a, b) -> Integer.compare(b.getValue(), a.getValue()))
                .limit(limit)
                .collect(ArrayList::new, (list, entry) -> list.add(entry), ArrayList::addAll);
        } finally {
            lock.unlock();
        }
    }
}

// Usage example
public static void main(String[] args) {
    PopularContentTracker tracker = new PopularContentTracker();

    tracker.increasePopularity(1);
    tracker.increasePopularity(2);
    tracker.increasePopularity(1);
    tracker.decreasePopularity(2);

    System.out.println("Most popular content: " + tracker.getMostPopularContent()); // Should return 1
    System.out.println("Content 1 popularity: " + tracker.getPopularity(1));
    System.out.println("Content 2 popularity: " + tracker.getPopularity(2));

    System.out.println("Top content:");
    tracker.getTopContent(5).forEach(entry ->
        System.out.println("Content " + entry.getKey() + ": " + entry.getValue())
    );
}
}

```

## Problem 5: Weighted Graph - Shortest Path

### Java Implementation with Dijkstra's Algorithm

java

```
import java.util.*;
```

```
public class WeightedGraph {  
    private final Map<String, Map<String, Integer>> graph;  
    private final Set<String> nodes;  
  
    public WeightedGraph() {  
        this.graph = new HashMap<>();  
        this.nodes = new HashSet<>();  
    }  
  
    public void addEdge(String source, String destination, int weight) {  
        graph.computeIfAbsent(source, k -> new HashMap<>()).put(destination, weight);  
        nodes.add(source);  
        nodes.add(destination);  
    }  
  
    public boolean canReach(String source, String destination) {  
        if (!nodes.contains(source) || !nodes.contains(destination)) {  
            return false;  
        }  
  
        if (source.equals(destination)) {  
            return true;  
        }  
  
        // BFS to check reachability  
        Set<String> visited = new HashSet<>();  
        Queue<String> queue = new LinkedList<>();  
        queue.offer(source);  
  
        while (!queue.isEmpty()) {  
            String current = queue.poll();  
  
            if (current.equals(destination)) {  
                return true;  
            }  
  
            if (visited.contains(current)) {  
                continue;  
            }  
  
            visited.add(current);  
  
            if (graph.containsKey(current)) {  
                for (String neighbor : graph.get(current).keySet()) {  
                    if (!visited.contains(neighbor)) {
```

```

        queue.offer(neighbor);
    }
}
}

return false;
}

public Optional<Integer> shortestPathTime(String source, String destination) {
    if (!nodes.contains(source) || !nodes.contains(destination)) {
        return Optional.empty();
    }

    if (source.equals(destination)) {
        return Optional.of(0);
    }

    // Dijkstra's algorithm
    Map<String, Integer> distances = new HashMap<>();
    PriorityQueue<NodeDistance> pq = new PriorityQueue<>(
        Comparator.comparingInt(nd -> nd.distance)
    );
    Set<String> visited = new HashSet<>();

    // Initialize distances
    for (String node : nodes) {
        distances.put(node, Integer.MAX_VALUE);
    }
    distances.put(source, 0);
    pq.offer(new NodeDistance(source, 0));

    while (!pq.isEmpty()) {
        NodeDistance current = pq.poll();
        String currentNode = current.node;
        int currentDist = current.distance;

        if (visited.contains(currentNode)) {
            continue;
        }

        visited.add(currentNode);

        if (currentNode.equals(destination)) {
            return Optional.of(currentDist);
        }

        // Check all neighbors
        if (graph.containsKey(currentNode)) {
            for (Map.Entry<String, Integer> edge : graph.get(currentNode).entrySet()) {

```

```

    for (Map.Entry<String, Integer> edge : graph.get(currentNode).entrySet()) {
        String neighbor = edge.getKey();
        int weight = edge.getValue();

        if (!visited.contains(neighbor)) {
            int newDistance = currentDist + weight;
            if (newDistance < distances.get(neighbor)) {
                distances.put(neighbor, newDistance);
                pq.offer(new NodeDistance(neighbor, newDistance));
            }
        }
    }
}

// Reconstruct path
if (distances.get(destination) == Integer.MAX_VALUE) {
    return new ShortestPathResult(Optional.empty(), Collections.emptyList());
}

List<String> path = new ArrayList<>();
String current = destination;
while (current != null) {
    path.add(current);
    current = previous.get(current);
}
Collections.reverse(path);

return new ShortestPathResult(Optional.of(distances.get(destination)), path);
}

// Helper classes
private static class NodeDistance {
    final String node;
    final int distance;

    NodeDistance(String node, int distance) {
        this.node = node;
        this.distance = distance;
    }
}

public static class ShortestPathResult {
    private final Optional<Integer> distance;
    private final List<String> path;

    public ShortestPathResult(Optional<Integer> distance, List<String> path) {
        this.distance = distance;
        this.path = new ArrayList<>(path);
    }
}

```

```

public Optional<Integer> getDistance() { return distance; }
public List<String> getPath() { return new ArrayList<>(path); }

@Override
public String toString() {
    return String.format("ShortestPathResult{distance=%s, path=%s}",
        distance.map(String::valueOf).orElse("No path"), path);
}
}

```

*// Advanced features*

```

public List<String> getAllNodes() {
    return new ArrayList<>(nodes);
}

```

```

public Map<String, Integer> getAllPaths(String source) {
    Map<String, Integer> allDistances = new HashMap<>();

    if (!nodes.contains(source)) {
        return allDistances;
    }
}

```

*// Run Dijkstra from source to all nodes*

```

Map<String, Integer> distances = new HashMap<>();
PriorityQueue<NodeDistance> pq = new PriorityQueue<>(
    Comparator.comparingInt(nd -> nd.distance)
);
Set<String> visited = new HashSet<>();

```

```

for (String node : nodes) {
    distances.put(node, Integer.MAX_VALUE);
}
distances.put(source, 0);
pq.offer(new NodeDistance(source, 0));

```

```

while (!pq.isEmpty()) {
    NodeDistance current = pq.poll();
    String currentNode = current.node;

    if (visited.contains(currentNode)) {
        continue;
    }
}

```

```

visited.add(currentNode);
allDistances.put(currentNode, distances.get(currentNode));

```

```

if (graph.containsKey(currentNode)) {
    for (Map.Entry<String, Integer> edge : graph.get(currentNode).entrySet()) {
        String nextNode = edge.getKey();
        Integer nextDistance = edge.getValue();
    }
}

```

```

        String neighbor = edge.getKey();
        int weight = edge.getValue();

        if (!visited.contains(neighbor)) {
            int newDistance = distances.get(currentNode) + weight;
            if (newDistance < distances.get(neighbor)) {
                distances.put(neighbor, newDistance);
                pq.offer(new NodeDistance(neighbor, newDistance));
            }
        }
    }
}

return allDistances;
}

// Usage example
public static void main(String[] args) {
    WeightedGraph graph = new WeightedGraph();
    graph.addEdge("A", "B", 100);
    graph.addEdge("B", "C", 200);
    graph.addEdge("A", "C", 350);
    graph.addEdge("C", "D", 150);

    System.out.println("Can reach A to D: " + graph.canReach("A", "D"));
    System.out.println("Shortest time A to D: " + graph.shortestPathTime("A", "D"));

    ShortestPathResult result = graph.getShortestPath("A", "D");
    System.out.println("Shortest path result: " + result);

    System.out.println("All distances from A:");
    graph.getAllPaths("A").forEach((node, distance) ->
        System.out.println(node + ": " + (distance == Integer.MAX_VALUE ? "unreachable" : distance + "ms"))
    );
}
}

```

## Problem 6: Job Interval Reporting

### Java Implementation for CI Pipeline Analysis

java

```
import java.util.*;
import java.util.stream.Collectors;

public class JobIntervalReporter {

    public static class Interval {
        private final int start;
        private final int end;

        public Interval(int start, int end) {
            this.start = start;
            this.end = end;
        }

        public int getStart() { return start; }
        public int getEnd() { return end; }

        public boolean overlaps(Interval other) {
            return this.start < other.end && other.start < this.end;
        }

        public Interval merge(Interval other) {
            return new Interval(Math.min(this.start, other.start),
                               Math.max(this.end, other.end));
        }

        public int duration() {
            return end - start;
        }

        @Override
        public String toString() {
            return String.format("[%d, %d]", start, end);
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            Interval interval = (Interval) obj;
            return start == interval.start && end == interval.end;
        }

        @Override
        public int hashCode() {
            return Objects.hash(start, end);
        }
    }
}
```



```

    }

    public List<Interval> mergeIntervals(List<Interval> intervals) {
        if (intervals == null || intervals.isEmpty()) {
            return new ArrayList<>();
        }

        // Sort intervals by start time
        List<Interval> sortedIntervals = intervals.stream()
            .sorted(Comparator.comparingInt(Interval::getStart))
            .collect(Collectors.toList());

        List<Interval> merged = new ArrayList<>();
        Interval current = sortedIntervals.get(0);

        for (int i = 1; i < sortedIntervals.size(); i++) {
            Interval next = sortedIntervals.get(i);

            if (current.getEnd() >= next.getStart()) {
                // Overlapping intervals - merge them
                current = current.merge(next);
            } else {
                // Non-overlapping - add current and move to next
                merged.add(current);
                current = next;
            }
        }

        merged.add(current); // Don't forget the last interval
        return merged;
    }

    public List<Interval> findActiveWindows(List<Interval> pipelineWindows) {
        return mergeIntervals(pipelineWindows);
    }

    public int getTotalActiveTime(List<Interval> pipelineWindows) {
        List<Interval> merged = mergeIntervals(pipelineWindows);
        return merged.stream()
            .mapToInt(Interval::duration)
            .sum();
    }

    public List<Interval> findIdlePeriods(List<Interval> pipelineWindows, Interval totalRange) {
        List<Interval> merged = mergeIntervals(pipelineWindows);
        List<Interval> idlePeriods = new ArrayList<>();

        if (merged.isEmpty()) {
            idlePeriods.add(totalRange);
            return idlePeriods;
        }
    }

```

```

    }

    // Check before first interval
    if (merged.get(0).getStart() > totalRange.getStart()) {
        idlePeriods.add(new Interval(totalRange.getStart(), merged.get(0).getStart()));
    }

    // Check between intervals
    for (int i = 0; i < merged.size() - 1; i++) {
        int gapStart = merged.get(i).getEnd();
        int gapEnd = merged.get(i + 1).getStart();
        if (gapStart < gapEnd) {
            idlePeriods.add(new Interval(gapStart, gapEnd));
        }
    }

    // Check after last interval
    Interval lastInterval = merged.get(merged.size() - 1);
    if (lastInterval.getEnd() < totalRange.getEnd()) {
        idlePeriods.add(new Interval(lastInterval.getEnd(), totalRange.getEnd()));
    }

    return idlePeriods;
}

public PeakConcurrencyResult findPeakConcurrencyTime(List<Interval> pipelineWindows) {
    if (pipelineWindows == null || pipelineWindows.isEmpty()) {
        return new PeakConcurrencyResult(0, Collections.emptyList());
    }

    List<TimeEvent> events = new ArrayList<>();

    // Create start and end events
    for (Interval interval : pipelineWindows) {
        events.add(new TimeEvent(interval.getStart(), 1)); // Start event
        events.add(new TimeEvent(interval.getEnd(), -1)); // End event
    }

    // Sort events by time, with end events before start events at same time
    events.sort((a, b) -> {
        if (a.time == b.time) {
            return Integer.compare(a.type, b.type); // -1 (end) before 1 (start)
        }
        return Integer.compare(a.time, b.time);
    });

    int currentCount = 0;
    int maxCount = 0;
    List<Interval> maxPeriods = new ArrayList<>();

```

```
Integer periodStart = null;
```

```
for (int i = 0; i < events.size(); i++) {
```

```
    int time = events.get(i).time;
```

```
    // Process all events at the same time
```

```
    while (i < events.size() && events.get(i).time == time) {
```

```
        currentCount += events.get(i).type;
```

```
        i++;
```

```
    }
```

```
    i--; // Adjust for the outer loop increment
```

```
    // Check if we have a new maximum
```

```
    if (currentCount > maxCount) {
```

```
        maxCount = currentCount;
```

```
        maxPeriods.clear();
```

```
        periodStart = time;
```

```
    }
```

```
    // If count drops below max, end the current period
```

```
    if (currentCount < maxCount && periodStart != null) {
```

```
        maxPeriods.add(new Interval(periodStart, time));
```

```
        periodStart = null;
```

```
    } else if (currentCount == maxCount && periodStart == null) {
```

```
        periodStart = time;
```

```
    }
```

```
}
```

```
return new PeakConcurrencyResult(maxCount, maxPeriods);
```

```
}
```

```
public PipelineEfficiencyReport analyzeEfficiency(List<Interval> pipelineWindows) {
```

```
    if (pipelineWindows == null || pipelineWindows.isEmpty()) {
```

```
        return new PipelineEfficiencyReport(0, 0, 0, 0.0, 0, 0.0);
```

```
    }
```

```
    int totalRuntime = pipelineWindows.stream()
```

```
        .mapToInt(Interval::duration)
```

```
        .sum();
```

```
    int activeTime = getTotalActiveTime(pipelineWindows);
```

```
    PeakConcurrencyResult peakResult = findPeakConcurrencyTime(pipelineWindows);
```

```
    double avgDuration = (double) totalRuntime / pipelineWindows.size();
```

```
    double efficiency = activeTime > 0 ? (double) activeTime / totalRuntime : 0.0;
```

```
    return new PipelineEfficiencyReport(
```

```
        pipelineWindows.size(),
```

```
        totalRuntime,
```

```
        activeTime,
```

```
        efficiency);
```

```
        efficiency,  
        peakResult.getMaxConcurrency(),  
        avgDuration  
    );  
}
```

*// Helper classes*

```
private static class TimeEvent {  
    final int time;  
    final int type; // 1 for start, -1 for end
```

```
    TimeEvent(int time, int type) {  
        this.time = time;  
        this.type = type;  
    }  
}
```

```
public static class PeakConcurrencyResult {  
    private final int maxConcurrency;  
    private final List<Interval> peakPeriods;
```

```
    public PeakConcurrencyResult(int maxConcurrency, List<Interval> peakPeriods) {  
        this.maxConcurrency = maxConcurrency;  
        this.peakPeriods = new ArrayList<>(peakPeriods);  
    }
```

```
    public int getMaxConcurrency() { return maxConcurrency; }  
    public List<Interval> getPeakPeriods() { return new ArrayList<>(peakPeriods); }
```

```
@Override  
    public String toString() {  
        return String.format("PeakConcurrencyResult{maxConcurrency=%d, peakPeriods=%s}",  
                               maxConcurrency, peakPeriods);  
    }  
}
```

```
public static class PipelineEfficiencyReport {  
    private final int totalPipelines;  
    private final int totalRuntime;  
    private final int activeTime;  
    private final double efficiency;  
    private final int maxConcurrency;  
    private final double avgPipelineDuration;
```

```
    public PipelineEfficiencyReport(int totalPipelines, int totalRuntime, int activeTime,  
                                    double efficiency, int maxConcurrency, double avgPipelineDuration) {  
        this.totalPipelines = totalPipelines;  
        this.totalRuntime = totalRuntime;  
        this.activeTime = activeTime;  
        this.efficiency = efficiency;
```

```

        this.maxConcurrency = maxConcurrency;
        this.avgPipelineDuration = avgPipelineDuration;
    }

    // Getters
    public int getTotalPipelines() { return totalPipelines; }
    public int getTotalRuntime() { return totalRuntime; }
    public int getActiveTime() { return activeTime; }
    public double getEfficiency() { return efficiency; }
    public int getMaxConcurrency() { return maxConcurrency; }
    public double getAvgPipelineDuration() { return avgPipelineDuration; }

```

```

@Override
public String toString() {
    return String.format(
        "PipelineEfficiencyReport{\n" +
        "  totalPipelines=%d,\n" +
        "  totalRuntime=%d,\n" +
        "  activeTime=%d,\n" +
        "  efficiency=%.2f%%,\n" +
        "  maxConcurrency=%d,\n" +
        "  avgPipelineDuration=%.2f\n" +
        "}",
        totalPipelines, totalRuntime, activeTime,
        efficiency * 100, maxConcurrency, avgPipelineDuration
    );
}
}

```

```

// Usage example
public static void main(String[] args) {
    JobIntervalReporter reporter = new JobIntervalReporter();

    // Example from the problem: [{2, 5}, {12, 15}, {4, 8}]
    List<Interval> pipelineWindows = Arrays.asList(
        new Interval(2, 5),
        new Interval(12, 15),
        new Interval(4, 8)
    );

    List<Interval> activeWindows = reporter.findActiveWindows(pipelineWindows);
    System.out.println("Active windows: " + activeWindows); // Output: [[2, 8], [12, 15]]

    int totalActive = reporter.getTotalActiveTime(pipelineWindows);
    System.out.println("Total active time: " + totalActive);

    PeakConcurrencyResult peakResult = reporter.findPeakConcurrencyTime(pipelineWindows);
    System.out.println("Peak concurrency: " + peakResult);
}

```

```
List<Interval> idlePeriods = reporter.findIdlePeriods(pipelineWindows, new Interval(0, 20));
System.out.println("Idle periods: " + idlePeriods);

PipelineEfficiencyReport efficiency = reporter.analyzeEfficiency(pipelineWindows);
System.out.println("Efficiency report:\n" + efficiency);
}
}
```

---

## Java-Specific Interview Preparation Topics

### 1. Core Java Concepts for Senior/Lead Roles

#### Essential Java Features:

java

*// Streams and Functional Programming*

```
public class StreamExamples {  
    public List<String> processEmployees(List<Employee> employees) {  
        return employees.stream()  
            .filter(emp -> emp.getDepartment().equals("Engineering"))  
            .filter(emp -> emp.getYearsExperience() > 5)  
            .sorted(Comparator.comparing(Employee::getName))  
            .map(Employee::getEmail)  
            .collect(Collectors.toList());  
    }  
}
```

*// Parallel processing for large datasets*

```
public Map<String, Long> getDepartmentCounts(List<Employee> employees) {  
    return employees.parallelStream()  
        .collect(Collectors.groupingBy(  
            Employee::getDepartment,  
            Collectors.counting()  
        ));  
}  
}
```

*// Advanced Collections Usage*

```
public class CollectionsExamples {  
    // Thread-safe collections  
    private final ConcurrentHashMap<String, AtomicInteger> counters = new ConcurrentHashMap<>();  
    private final CopyOnWriteArrayList<String> auditLog = new CopyOnWriteArrayList<>();  
  
    // Custom comparators and sorting  
    public void sortEmployeesByMultipleCriteria(List<Employee> employees) {  
        employees.sort(  
            Comparator.comparing(Employee::getDepartment)  
                .thenComparing(Employee::getLevel)  
                .thenComparing(Employee::getName)  
        );  
    }  
}
```

*// Memory Management and Performance*

```
public class MemoryOptimization {  
    // Use primitive collections to avoid boxing overhead  
    private final TIntObjectHashMap<String> idToName = new TIntObjectHashMap<>();  
  
    // Object pooling for frequent allocations  
    private final ObjectPool<StringBuilder> stringBuilderPool =  
        new GenericObjectPool<>(new StringBuilderFactory());  
  
    // Weak references for caches
```

```
private final WeakHashMap<String, ExpensiveObject> cache = new WeakHashMap<>();  
}
```

## 2. Concurrency and Multithreading



java

*// Advanced concurrency patterns*

public class ConcurrencyPatterns {

*// Producer-Consumer with BlockingQueue*

public class TaskProcessor {

private final BlockingQueue<Task> taskQueue = new ArrayBlockingQueue<>(1000);

private final ExecutorService executorService = Executors.newFixedThreadPool(10);

public void startProcessing() {

for (int i = 0; i < 5; i++) {

executorService.submit(this::processTasks);

}

}

private void processTasks() {

while (!Thread.currentThread().isInterrupted()) {

try {

Task task = taskQueue.take();

processTask(task);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

break;

}

}

}

}

*// Lock-free data structures*

public class LockFreeCounter {

private final AtomicLong counter = new AtomicLong(0);

public long increment() {

return counter.incrementAndGet();

}

public long get() {

return counter.get();

}

}

*// CompletableFuture for async processing*

public class AsyncProcessor {

public CompletableFuture<String> processDataAsync(String input) {

return CompletableFuture

.supplyAsync(() -> heavyComputation(input))

.thenApply(this::transform)

.thenCompose(this::validate)

```

        .exceptionally(this::handleError);
    }

    public CompletableFuture<List<String>> processMultipleAsync(List<String> inputs) {
        List<CompletableFuture<String>> futures = inputs.stream()
            .map(this::processDataAsync)
            .collect(Collectors.toList());

        return CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .collect(Collectors.toList()));
    }
}
}
}

```

### 3. Spring Framework & Enterprise Java

java

*// Spring Boot application structure*

@SpringBootApplication

@EnableAsync

@EnableScheduling

public class AtlassianStyleApplication {

    @Bean

    @Primary

    public TaskExecutor taskExecutor() {

        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

        executor.setCorePoolSize(10);

        executor.setMaxPoolSize(50);

        executor.setQueueCapacity(100);

        executor.setThreadNamePrefix("async-task-");

        executor.initialize();

        return executor;

    }

}

*// Service layer with proper error handling*

@Service

@Transactional

public class EmployeeService {

    private final EmployeeRepository employeeRepository;

    private final NotificationService notificationService;

    private final CacheManager cacheManager;

    public EmployeeService(EmployeeRepository employeeRepository,

        NotificationService notificationService,

        CacheManager cacheManager) {

        this.employeeRepository = employeeRepository;

        this.notificationService = notificationService;

        this.cacheManager = cacheManager;

    }

    @Cacheable(value = "employees", key = "#id")

    public Optional<Employee> findById(Long id) {

        return employeeRepository.findById(id);

    }

    @Async

    @Retryable(value = {Exception.class}, maxAttempts = 3, backoff = @Backoff(delay = 1000))

    public CompletableFuture<Void> processEmployeeAsync(Long employeeId) {

        try {

            Employee employee = findById(employeeId)

                .orElseThrow(() -> new EmployeeNotFoundException("Employee not found: " + employeeId));

```

// Heavy processing
processEmployee(employee);

// Send notification
notificationService.sendNotification(employee);

return CompletableFuture.completedFuture(null);
} catch (Exception e) {
    log.error("Error processing employee: {}", employeeId, e);
    throw e;
}
}

@EventListener
public void handleEmployeeCreated(EmployeeCreatedEvent event) {
    // Clear related caches
    cacheManager.getCache("departments").evict(event.getEmployee().getDepartmentId());
}
}

// REST API with proper validation and error handling
@RestController
@RequestMapping("/api/v1/employees")
@Validated
public class EmployeeController {

    private final EmployeeService employeeService;

    @GetMapping("/{id}")
    public ResponseEntity<EmployeeDto> getEmployee(
        @PathVariable @Min(1) Long id,
        HttpServletRequest request) {

        Optional<Employee> employee = employeeService.findById(id);

        if (employee.isEmpty()) {
            return ResponseEntity.notFound().build();
        }

        EmployeeDto dto = EmployeeMapper.toDto(employee.get());

        // Add HATEOAS links
        dto.add(linkTo(methodOn(EmployeeController.class).getEmployee(id)).withSelfRel());
        dto.add(linkTo(methodOn(DepartmentController.class)
            .getDepartment(employee.get().getDepartmentId())).withRel("department"));

        return ResponseEntity.ok()
            .cacheControl(CacheControl.maxAge(Duration.ofMinutes(5)))
            .body(dto);
    }
}

```

```

        .body(dto)
    }

    @PostMapping
    public ResponseEntity<EmployeeDto> createEmployee(
        @RequestBody @Valid CreateEmployeeRequest request,
        UriComponentsBuilder uriBuilder) {

        Employee employee = employeeService.createEmployee(request);
        EmployeeDto dto = EmployeeMapper.toDto(employee);

        URI location = uriBuilder.path("/api/v1/employees/{id}")
            .buildAndExpand(employee.getId())
            .toUri();

        return ResponseEntity.created(location).body(dto);
    }
}

```

## 4. Database & JPA Best Practices

java

*// Optimized JPA entities*

@Entity

@Table(name = "employees", indexes = {

    @Index(name = "idx\_employee\_email", columnList = "email"),

    @Index(name = "idx\_employee\_dept\_status", columnList = "department\_id, status")

})

@EntityListeners(AuditingEntityListener.class)

public class Employee {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    @Column(nullable = false, unique = true)

    private String email;

    @ManyToOne(fetch = FetchType.LAZY)

    @JoinColumn(name = "department\_id")

    private Department department;

    @OneToMany(mappedBy = "employee", cascade = CascadeType.ALL, orphanRemoval = true)

    @BatchSize(size = 20)

    private List<Project> projects = new ArrayList<>();

    @CreatedDate

    private LocalDateTime createdAt;

    @LastModifiedDate

    private LocalDateTime updatedAt;

    @Version

    private Long version;

}

*// Repository with custom queries*

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long>, JpaSpecificationExecutor<Employee> {

    @Query("SELECT e FROM Employee e JOIN FETCH e.department WHERE e.status = :status")

    List<Employee> findActiveEmployeesWithDepartment(@Param("status") EmployeeStatus status);

    @Query(value = "SELECT \* FROM employees e WHERE e.hire\_date >= :fromDate",

        countQuery = "SELECT count(\*) FROM employees e WHERE e.hire\_date >= :fromDate",

        nativeQuery = true)

    Page<Employee> findRecentHires(@Param("fromDate") LocalDate fromDate, Pageable pageable);

    @Modifying

```

@Query("UPDATE Employee e SET e.status = :newStatus WHERE e.department.id = :deptId")
int bulkUpdateEmployeeStatus(@Param("deptId") Long departmentId,
                             @Param("newStatus") EmployeeStatus newStatus);
}

// Database migration management
@Component
public class DatabaseMigrations {

    @EventListener(ApplicationReadyEvent.class)
    public void performDataMigration() {
        // Custom data migration logic
        migrateOldEmployeeData();
    }

    @Transactional
    public void migrateOldEmployeeData() {
        // Batch processing for large datasets
        int batchSize = 1000;
        int offset = 0;

        List<OldEmployee> batch;
        do {
            batch = oldEmployeeRepository.findBatch(offset, batchSize);

            List<Employee> newEmployees = batch.stream()
                .map(this::convertToNewFormat)
                .collect(Collectors.toList());

            employeeRepository.saveAll(newEmployees);

            offset += batchSize;
        } while (!batch.isEmpty());
    }
}

```

## 5. Testing Strategies for Java

java

*// Unit testing with Mockito and JUnit 5*

@ExtendWith(MockitoExtension.class)

class EmployeeServiceTest {

@Mock

private EmployeeRepository employeeRepository;

@Mock

private NotificationService notificationService;

@InjectMocks

private EmployeeService employeeService;

@Test

void shouldCreateEmployeeSuccessfully() {

*// Given*

CreateEmployeeRequest request = new CreateEmployeeRequest("john@example.com", "Engineering");

Employee savedEmployee = new Employee();

savedEmployee.setId(1L);

savedEmployee.setEmail("john@example.com");

when(employeeRepository.save(any(Employee.class))).thenReturn(savedEmployee);

*// When*

Employee result = employeeService.createEmployee(request);

*// Then*

assertThat(result.getId()).isEqualTo(1L);

assertThat(result.getEmail()).isEqualTo("john@example.com");

verify(employeeRepository).save(argThat(emp ->

emp.getEmail().equals("john@example.com"));

verify(notificationService).sendWelcomeEmail(savedEmployee);

}

@Test

void shouldThrowExceptionWhenEmployeeNotFound() {

*// Given*

when(employeeRepository.findById(999L)).thenReturn(Optional.empty());

*// When & Then*

assertThatThrownBy(() -> employeeService.getEmployee(999L))

.isInstanceOf(EmployeeNotFoundException.class)

.hasMessageContaining("Employee not found: 999");

}

}



// Integration testing with TestContainers

@SpringBootTest

@Testcontainers

class EmployeeIntegrationTest {

@Container

```
static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:13")
    .withDatabaseName("testdb")
    .withUsername("test")
    .withPassword("test");
```

@DynamicPropertySource

```
static void configureProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url", postgres::getJdbcUrl);
    registry.add("spring.datasource.username", postgres::getUsername);
    registry.add("spring.datasource.password", postgres::getPassword);
}
```

@Autowired

private TestRestTemplate restTemplate;

@Autowired

private EmployeeRepository employeeRepository;

@Test

void shouldCreateAndRetrieveEmployee() {

*// Given*

CreateEmployeeRequest request = new CreateEmployeeRequest("jane@example.com", "Engineering");

*// When - Create employee*

ResponseEntity<EmployeeDto> createResponse = restTemplate.postForEntity(
 "/api/v1/employees", request, EmployeeDto.class);

*// Then - Verify creation*

assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);
EmployeeDto createdEmployee = createResponse.getBody();
assertThat(createdEmployee.getEmail()).isEqualTo("jane@example.com");

*// When - Retrieve employee*

ResponseEntity<EmployeeDto> getResponse = restTemplate.getForEntity(
 "/api/v1/employees/" + createdEmployee.getId(), EmployeeDto.class);

*// Then - Verify retrieval*

assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
EmployeeDto retrievedEmployee = getResponse.getBody();
assertThat(retrievedEmployee.getEmail()).isEqualTo("jane@example.com");

}

}

// Performance testing

```
// Performance testing
@Test
@Timeout(value = 5, unit = TimeUnit.SECONDS)
void shouldHandleLargeDatasetEfficiently() {
    // Test with large dataset
    List<Employee> employees = generateTestEmployees(10000);

    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start();

    List<EmployeeDto> result = employeeService.processLargeDataset(employees);

    stopWatch.stop();

    assertThat(result).hasSize(10000);
    assertThat(stopWatch.getTotalTimeMillis()).isLessThan(3000);
}
```

## 6. Design Patterns for Enterprise Java

java

*// Factory Pattern for different employee types*

```
public class EmployeeFactory {

    public static Employee createEmployee(EmployeeType type, String name, String email) {
        switch (type) {
            case ENGINEER:
                return new Engineer(name, email);
            case MANAGER:
                return new Manager(name, email);
            case DESIGNER:
                return new Designer(name, email);
            default:
                throw new IllegalArgumentException("Unknown employee type: " + type);
        }
    }
}
```

*// Strategy Pattern for different notification methods*

```
public interface NotificationStrategy {
    void sendNotification(String message, String recipient);
}
```

@Component("emailNotification")

```
public class EmailNotificationStrategy implements NotificationStrategy {
    @Override
    public void sendNotification(String message, String recipient) {
        // Email implementation
    }
}
```

@Component("slackNotification")

```
public class SlackNotificationStrategy implements NotificationStrategy {
    @Override
    public void sendNotification(String message, String recipient) {
        // Slack implementation
    }
}
```

@Service

```
public class NotificationService {
    private final Map<String, NotificationStrategy> strategies;

    public NotificationService(Map<String, NotificationStrategy> strategies) {
        this.strategies = strategies;
    }

    public void sendNotification(String type, String message, String recipient) {
```

```

NotificationStrategy strategy = strategies.get(type + "Notification");
if (strategy != null) {
    strategy.sendNotification(message, recipient);
}
}
}

```

*// Observer Pattern for event handling*

@Component

```

public class EmployeeEventPublisher {
    private final ApplicationEventPublisher eventPublisher;

    public EmployeeEventPublisher(ApplicationEventPublisher eventPublisher) {
        this.eventPublisher = eventPublisher;
    }

    public void publishEmployeeCreated(Employee employee) {
        EmployeeCreatedEvent event = new EmployeeCreatedEvent(this, employee);
        eventPublisher.publishEvent(event);
    }
}

```

@EventListener

@Component

```

public class EmployeeEventHandler {

    @Async
    @EventListener
    public void handleEmployeeCreated(EmployeeCreatedEvent event) {
        // Send welcome email
        sendWelcomeEmail(event.getEmployee());
    }
}

```

@EventListener

```

public void handleEmployeeCreated(EmployeeCreatedEvent event) {
    // Update analytics
    updateEmployeeAnalytics(event.getEmployee());
}
}

```

*// Command Pattern for undo/redo operations (like Jira)*

```

public interface Command {
    void execute();
    void undo();
}

public class UpdateEmployeeCommand implements Command {
    private final Employee employee;
    private final String field;
    private final Object newValue;
}

```

```

private final Object newValue;
private Object oldValue;

public UpdateEmployeeCommand(Employee employee, String field, Object newValue) {
    this.employee = employee;
    this.field = field;
    this.newValue = newValue;
}

@Override
public void execute() {
    this.oldValue = getFieldValue(employee, field);
    setFieldValue(employee, field, newValue);
}

@Override
public void undo() {
    setFieldValue(employee, field, oldValue);
}
}

@Service
public class CommandManager {
    private final Stack<Command> commandHistory = new Stack<>();
    private final Stack<Command> redoStack = new Stack<>();

    public void executeCommand(Command command) {
        command.execute();
        commandHistory.push(command);
        redoStack.clear(); // Clear redo stack when new command is executed
    }

    public void undo() {
        if (!commandHistory.isEmpty()) {
            Command command = commandHistory.pop();
            command.undo();
            redoStack.push(command);
        }
    }

    public void redo() {
        if (!redoStack.isEmpty()) {
            Command command = redoStack.pop();
            command.execute();
            commandHistory.push(command);
        }
    }
}

```

## 7. Microservices & Cloud-Native Java

java

*// Service discovery with Spring Cloud*

@RestController

@RefreshScope *// For dynamic config updates*

public class EmployeeController {

@Value("\${employee.service.max-results:100}")

private int maxResults;

@Autowired

private DiscoveryClient discoveryClient;

@Autowired

@LoadBalanced

private RestTemplate restTemplate;

@GetMapping("/employees/{id}/projects")

public ResponseEntity<List<ProjectDto>> getEmployeeProjects(@PathVariable Long id) {

*// Call another microservice using service discovery*

String projectServiceUrl = "http://project-service/api/v1/projects/employee/" + id;

try {

ProjectDto[] projects = restTemplate.getForObject(projectServiceUrl, ProjectDto[].class);

return ResponseEntity.ok(Arrays.asList(projects));

} catch (Exception e) {

*// Fallback mechanism*

return ResponseEntity.ok(getProjectsFallback(id));

}

}

@CircuitBreaker(name = "project-service", fallbackMethod = "getProjectsFallback")

@TimeLimiter(name = "project-service")

@Retry(name = "project-service")

public CompletableFuture<List<ProjectDto>> getEmployeeProjectsAsync(Long id) {

return CompletableFuture.supplyAsync(() -> {

String url = "http://project-service/api/v1/projects/employee/" + id;

ProjectDto[] projects = restTemplate.getForObject(url, ProjectDto[].class);

return Arrays.asList(projects);

});

}

public CompletableFuture<List<ProjectDto>> getProjectsFallback(Long id, Exception ex) {

log.warn("Fallback triggered for employee {}: {}", id, ex.getMessage());

return CompletableFuture.completedFuture(Collections.emptyList());

}

}

*// Health checks and metrics*

@Component

```
public class EmployeeServiceHealthIndicator implements HealthIndicator {
```

```
    private final EmployeeRepository employeeRepository;
```

```
    @Override
```

```
    public Health health() {
```

```
        try {
```

```
            long count = employeeRepository.count();
```

```
            return Health.up()
```

```
                .withDetail("employee-count", count)
```

```
                .withDetail("status", "Service is running normally")
```

```
                .build();
```

```
        } catch (Exception e) {
```

```
            return Health.down()
```

```
                .withDetail("error", e.getMessage())
```

```
                .build();
```

```
        }
```

```
    }
```

```
}
```

```
// Custom metrics
```

```
@Component
```

```
public class EmployeeMetrics {
```

```
    private final Counter employeeCreationCounter;
```

```
    private final Timer employeeProcessingTimer;
```

```
    private final Gauge activeEmployeesGauge;
```

```
    public EmployeeMetrics(MeterRegistry meterRegistry, EmployeeRepository employeeRepository) {
```

```
        this.employeeCreationCounter = Counter.builder("employees.created")
```

```
            .description("Number of employees created")
```

```
            .register(meterRegistry);
```

```
        this.employeeProcessingTimer = Timer.builder("employees.processing.time")
```

```
            .description("Employee processing time")
```

```
            .register(meterRegistry);
```

```
        this.activeEmployeesGauge = Gauge.builder("employees.active")
```

```
            .description("Number of active employees")
```

```
            .register(meterRegistry, this, EmployeeMetrics::getActiveEmployeeCount);
```

```
    }
```

```
    public void incrementEmployeeCreation() {
```

```
        employeeCreationCounter.increment();
```

```
    }
```

```
    public Timer.Sample startProcessingTimer() {
```

```
        return Timer.start();
```

```
    }
```

```
}  
  
private double getActiveEmployeeCount() {  
    // This will be called periodically to update the gauge  
    return employeeRepository.countByStatus(EmployeeStatus.ACTIVE);  
}  
}
```

## 8. Security Best Practices



java

*// JWT Security Configuration*

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class SecurityConfig {

    @Bean

    public PasswordEncoder passwordEncoder() {

        return new BCryptPasswordEncoder(12);

    }

    @Bean

    public JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint() {

        return new JwtAuthenticationEntryPoint();

    }

    @Bean

    public JwtRequestFilter jwtRequestFilter() {

        return new JwtRequestFilter();

    }

    @Bean

    protected SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        http.csrf().disable()

        .authorizeHttpRequests(authz -> authz

            .requestMatchers("/api/v1/auth/\*\*").permitAll()

            .requestMatchers("/actuator/health").permitAll()

            .requestMatchers(HttpMethod.GET, "/api/v1/employees/\*\*").hasAnyRole("USER", "ADMIN")

            .requestMatchers(HttpMethod.POST, "/api/v1/employees/\*\*").hasRole("ADMIN")

            .anyRequest().authenticated()

        )

        .exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint())

        .and()

        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

        .and()

        .addFilterBefore(jwtRequestFilter(), UsernamePasswordAuthenticationFilter.class);

        return http.build();

    }

}

*// Input validation and sanitization*

@RestController

@Validated

public class SecureEmployeeController {

    @PostMapping("/employees")

```

public ResponseEntity<EmployeeDto> createEmployee(
    @RequestBody @Valid CreateEmployeeRequest request,
    @AuthenticationPrincipal UserPrincipal currentUser) {

    // Validate input
    if (!isValidEmail(request.getEmail())) {
        throw new InvalidInputException("Invalid email format");
    }

    // Authorization check
    if (!hasPermissionToCreateEmployee(currentUser, request.getDepartment())) {
        throw new AccessDeniedException("Insufficient permissions");
    }

    // Sanitize input
    String sanitizedName = sanitizeInput(request.getName());
    String sanitizedEmail = sanitizeEmail(request.getEmail());

    CreateEmployeeRequest sanitizedRequest = new CreateEmployeeRequest(
        sanitizedName, sanitizedEmail, request.getDepartment());

    Employee employee = employeeService.createEmployee(sanitizedRequest);
    return ResponseEntity.ok(EmployeeMapper.toDto(employee));
}

@PreAuthorize("hasRole('ADMIN') or #employeeId == authentication.principal.employeeId")
@GetMapping("/employees/{employeeId}")
public ResponseEntity<EmployeeDto> getEmployee(@PathVariable Long employeeId) {
    // Method-level security handled by @PreAuthorize
    return employeeService.findById(employeeId)
        .map(EmployeeMapper::toDto)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}
}

```

## 9. Performance Optimization Techniques

java

*// Caching strategies*

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

public CacheManager cacheManager() {

RedisCacheManager.Builder builder = RedisCacheManager

.RedisCacheManagerBuilder

.fromConnectionFactory(redisConnectionFactory())

.cacheDefaults(cacheConfiguration());

return builder.build();

}

private RedisCacheConfiguration cacheConfiguration() {

return RedisCacheConfiguration.defaultCacheConfig()

.entryTtl(Duration.ofMinutes(10))

.serializeKeysWith(RedisSerializationContext.SerializationPair

.fromSerializer(new StringRedisSerializer()))

.serializeValuesWith(RedisSerializationContext.SerializationPair

.fromSerializer(new GenericJackson2JsonRedisSerializer()));

}

}

*// Database optimization*

@Service

public class OptimizedEmployeeService {

*// Use projections to fetch only needed data*

@Query("SELECT new com.atlassian.dto.EmployeeSummary(e.id, e.name, e.email) " +

"FROM Employee e WHERE e.department.id = :deptId")

List<EmployeeSummary> findEmployeeSummariesByDepartment(@Param("deptId") Long deptId);

*// Batch operations*

@Transactional

public void updateEmployeesBatch(List<Long> employeeIds, EmployeeStatus newStatus) {

int batchSize = 1000;

for (int i = 0; i < employeeIds.size(); i += batchSize) {

int end = Math.min(i + batchSize, employeeIds.size());

List<Long> batch = employeeIds.subList(i, end);

employeeRepository.updateStatusBatch(batch, newStatus);

*// Clear persistence context periodically to avoid memory issues*

entityManager.flush();

entityManager.clear();

}

```

    }

    // Async processing for heavy operations
    @Async("taskExecutor")
    @Transactional
    public CompletableFuture<Void> processEmployeeDataAsync(List<Employee> employees) {
        return CompletableFuture.runAsync() -> {
            employees.parallelStream().forEach(this::processHeavyOperation);
        };
    }
}

// Memory optimization
@Component
public class MemoryOptimizedProcessor {

    // Use primitive collections to avoid boxing
    private final TLongObjectHashMap<String> employeeIdToEmail = new TLongObjectHashMap<>();

    // Object pooling for frequently created objects
    private final GenericObjectPool<StringBuilder> stringBuilderPool;

    public MemoryOptimizedProcessor() {
        GenericObjectPoolConfig<StringBuilder> config = new GenericObjectPoolConfig<>();
        config.setMaxTotal(100);
        config.setMaxIdle(50);
        config.setMinIdle(10);

        this.stringBuilderPool = new GenericObjectPool<>(
            new BasePooledObjectFactory<StringBuilder>() {
                @Override
                public StringBuilder create() {
                    return new StringBuilder(256);
                }

                @Override
                public PooledObject<StringBuilder> wrap(StringBuilder obj) {
                    obj.setLength(0); // Reset the StringBuilder
                    return new DefaultPooledObject<>(obj);
                }
            }, config);
    }

    public String processLargeText(String input) {
        StringBuilder sb = null;
        try {
            sb = stringBuilderPool.borrowObject();
            // Use the StringBuilder for processing
            sb.append(input).append(" processed");
            return sb.toString();
        }
    }
}

```

```

    return sb.toString();
} catch (Exception e) {
    throw new RuntimeException("Error processing text", e);
} finally {
    if (sb != null) {
        try {
            stringBuilderPool.returnObject(sb);
        } catch (Exception e) {
            log.warn("Error returning StringBuilder to pool", e);
        }
    }
}
}
}
}
}

```

## 10. Mock Interview Questions - Java Specific

### Technical Questions:

1. **"Design a real-time collaborative editing system like Confluence using Java"**
  - Focus on WebSockets, operational transformation, conflict resolution
  - Use Spring WebSocket, Redis for pub/sub, optimistic locking
2. **"Implement a workflow engine similar to Jira's status transitions"**
  - State machine pattern, validation rules, audit trails
  - Use Spring State Machine, JPA auditing
3. **"Design a plugin architecture for Atlassian products"**
  - OSGi bundles, class loading isolation, dependency injection
  - Spring Boot with custom class loaders
4. **"Create a distributed caching system for high-performance APIs"**
  - Redis clustering, cache invalidation strategies, fallback mechanisms
  - Spring Cache with Redis, circuit breakers
5. **"Implement rate limiting with different strategies"**
  - Token bucket, sliding window, distributed rate limiting
  - Redis-based counters, Spring AOP for cross-cutting concerns

### Behavioral Questions:

1. **"How do you handle technical debt in a Java codebase?"**
2. **"Describe your approach to code reviews and mentoring junior Java developers"**
3. **"How do you ensure thread safety in a multi-threaded Java application?"**
4. **"What's your strategy for optimizing JVM performance in production?"**
5. **"How do you approach testing in a microservices architecture?"**

## 11. Java-Specific Preparation Checklist

### Core Java (2-3 hours daily):

- ☐ Collections framework and concurrent collections
- ☐ Multithreading, locks, and concurrent utilities
- ☐ JVM internals, garbage collection, memory management
- ☐ Stream API and functional programming
- ☐ Exception handling best practices

### Spring Ecosystem (2-3 hours daily):

- ☐ Spring Boot auto-configuration and starters
- ☐ Spring Security for authentication/authorization
- ☐ Spring Data JPA with query optimization
- ☐ Spring Cloud for microservices
- ☐ Spring WebFlux for reactive programming

### Enterprise Patterns (1-2 hours daily):

- ☐ Design patterns (Factory, Strategy, Observer, Command)
- ☐ Architectural patterns (MVC, Repository, Service Layer)
- ☐ Integration patterns (Message queues, Event sourcing)
- ☐ Caching strategies and implementation

### Testing (1 hour daily):

- ☐ JUnit 5 and TestNG
- ☐ Mockito for mocking
- ☐ TestContainers for integration testing
- ☐ Performance testing with JMH

### Tools & Build Systems:

- ☐ Maven/Gradle build optimization
- ☐ Docker containerization
- ☐ CI/CD with Jenkins/GitLab CI
- ☐ Monitoring with Micrometer/Actuator

### Final Java Interview Tips:

1. **Code Quality:** Always write clean, well-commented code during interviews
2. **Exception Handling:** Demonstrate proper exception handling and logging
3. **Thread Safety:** Show understanding of concurrent programming concepts
4. **Performance:** Discuss time/space complexity and optimization strategies
5. **Testing:** Write testable code and discuss testing strategies
6. **Best Practices:** Follow Java naming conventions and coding standards
7. **Framework Knowledge:** Be prepared to discuss Spring Boot internals
8. **Microservices:** Understand distributed system challenges and solutions

Remember to practice coding problems on platforms like LeetCode, HackerRank, and CodeSignal with Java. Focus on implementing solutions that demonstrate enterprise-level thinking and best practices that would be expected in a lead role at Atlassian.

```
new NodeDistance(neighbor, newDistance));  
}  
}  
}  
}  
}
```

```

return Optional.empty(); // No path found
}

public ShortestPathResult getShortestPath(String source, String destination) {
    if (!nodes.contains(source) || !nodes.contains(destination)) {
        return new ShortestPathResult(Optional.empty(), Collections.emptyList());
    }

    if (source.equals(destination)) {
        return new ShortestPathResult(Optional.of(0), Arrays.asList(source));
    }

    Map<String, Integer> distances = new HashMap<>();
    Map<String, String> previous = new HashMap<>();
    PriorityQueue<NodeDistance> pq = new PriorityQueue<>(
        Comparator.comparingInt(nd -> nd.distance)
    );
    Set<String> visited = new HashSet<>();

    // Initialize
    for (String node : nodes) {
        distances.put(node, Integer.MAX_VALUE);
        previous.put(node, null);
    }
    distances.put(source, 0);
    pq.offer(new NodeDistance(source, 0));

    while (!pq.isEmpty()) {
        NodeDistance current = pq.poll();
        String currentNode = current.node;

        if (visited.contains(currentNode)) {
            continue;
        }

        visited.add(currentNode);

        if (graph.containsKey(currentNode)) {
            for (Map.Entry<String, Integer> edge : graph.get(currentNode).entrySet()) {
                String neighbor = edge.getKey();
                int weight = edge.getValue();

                if (!visited.contains(neighbor)) {
                    int newDistance = distances.get(currentNode) + weight;
                    if (newDistance < distances.get(neighbor)) {
                        distances.put(neighbor, newDistance);
                        previous.put(neighbor, currentNode);
                        pq.offer(

```



