1. **Queue + Exception Handling + OOP**
   Design a **ticket booking system** where customers are added to a queue for processing. Implement a method to serve the next customer and remove them from the queue. Handle exceptions if the queue is empty.
2. **PriorityQueue + OOP + Exception Handling**
   Create a **task scheduling system** using `PriorityQueue`. Tasks should be prioritized based on urgency (e.g., High, Medium, Low). Implement a method to retrieve and execute tasks in order of priority. Handle exceptions for invalid priority levels.
3. **Queue + Factory Pattern + Exception Handling**
   Implement a **notification management system** using a queue. Design a factory class to create different types of notifications (`SMSNotification`, `EmailNotification`). Use a queue to store and send notifications. Handle exceptions if the queue is empty.
4. **Queue + Iterator Pattern + OOP**
   Create a **print queue management system** where documents are added to a `LinkedList` queue. Implement a custom iterator to traverse through the documents in the order they were added. Provide methods to remove completed print jobs.
5. **Queue + State Management + Exception Handling**
   Design a **food delivery management system** using a queue to store delivery orders. Each order has a state (`Placed`, `Prepared`, `Delivered`). Implement a method to update the state and handle exceptions for invalid state transitions.
6. **Deque + OOP + Exception Handling**
   Develop a **recent activity tracker** using `Deque`. Store the most recent 10 activities and display them in reverse order using the `Deque` interface. Implement exception handling to prevent accessing elements when the deque is empty.
7. **BlockingQueue + Exception Handling + Multithreading**
   Create a **producer-consumer system** using `BlockingQueue`. Producers should add tasks to the queue, and consumers should process them. Implement exception handling for interrupted threads and ensure graceful shutdown.
8. **Queue + Strategy Pattern + OOP**
   Implement a **task execution system** using a queue where tasks are executed using different strategies (e.g., `SequentialExecution`, `ParallelExecution`). Use the strategy pattern to switch execution methods. Handle exceptions for failed tasks.
9. **Queue + Generics + Exception Handling**
   Create a generic class `MessageQueue<T>` that can store messages of any type using a `Queue`. Implement methods to add, retrieve, and process messages. Ensure exceptions are handled when retrieving from an empty queue.
10. **Queue + MVC Pattern + OOP**
    Develop a small part of a **customer service system** using the MVC pattern. The model should store customer queries using a `Queue`, the controller should manage query assignment, and the view should display current queries. Implement exceptions for handling empty queues.
11. **Deque + Factory Pattern + Exception Handling**
    Create a **document versioning system** where each version of a document is stored using a `Deque`. Implement methods to save, retrieve, and undo changes using `Deque` operations. Handle exceptions for undoing when no previous versions exist.
12. **Queue + Composite Pattern + OOP**
    Design a **workflow management system** using a queue to manage tasks. Some tasks can be composite tasks (containing subtasks). Implement methods to execute subtasks in sequence. Ensure exceptions are handled for invalid or incomplete tasks.

13. **Queue + Real-time Monitoring + Exception Handling**
    Build a basic part of a **traffic monitoring system** where vehicle entry data is stored using a queue. Implement methods to simulate vehicle exit using FIFO logic. Handle exceptions if the queue is empty when a vehicle attempts to exit.
14. **PriorityQueue + OOP + Exception Handling**
    Develop an **emergency response system** where emergencies are stored using `PriorityQueue` based on severity. Implement a method to retrieve and respond to the most critical emergency. Handle exceptions for empty queues.
15. **Deque + OOP + Exception Handling**
    Create a **web browser session** using a `Deque` to manage the back and forward history. Implement methods to navigate back and forward between pages. Handle exceptions for edge cases like navigating back without any previous pages.
16. **Collections + Exception Handling + OOP**
    Design a part of an inventory management system using a `HashMap` to store product details (product ID as key, product name as value). Implement a method to retrieve a product by its ID. Handle exceptions using a custom exception if the product is not found.
17. **Generics + Collections + OOP**
    Create a generic class `DataStore<T>` that uses an `ArrayList` to store objects of any type. Implement methods to add, remove, and search for objects. Ensure it throws appropriate exceptions for invalid operations.
18. **Factory Pattern + Exception Handling + Collections**
    Implement a simple factory class to create different types of lists (`ArrayList`, `LinkedList`). Provide a method to print all elements from the list using a common interface. Handle exceptions for unsupported operations.
19. **State Management + Collections + Exception Handling**
    Design a state management feature for a ticket booking system using an `Enum` to represent states (`BOOKED`, `CANCELLED`, `CONFIRMED`). Store booking details using a `HashMap`. Implement exception handling for invalid state transitions.
20. **Iterator Pattern + OOP + Collections**
    Implement an iterator for a `LinkedList` storing customer data. The iterator should traverse and print all customer names. Also, provide a method to reset the iterator. Handle exceptions when trying to iterate an empty list.
21. **Exception Handling + Collections + Decorator Pattern**
    Develop a logging feature using the decorator pattern. Create a base logger and extend it with additional functionalities (e.g., timestamp logging, error level logging). Store logs using an `ArrayList`. Handle exceptions for invalid log entries.
22. **Strategy Pattern + Collections + Exception Handling**
    Implement a sorting module using the strategy pattern. Provide multiple sorting strategies (e.g., Bubble Sort, Quick Sort) for sorting an `ArrayList` of integers. Implement exception handling for empty or null lists.
23. **OOP + Exception Handling + Collections**
    Develop a user authentication system using a `HashMap` to store username-password pairs. Implement a method to validate user credentials. Throw appropriate exceptions for invalid usernames or incorrect passwords.
24. **Factory Pattern + OOP + Collections**
    Design a system where a factory class generates different types of vehicles (`Car`, `Bike`). Use a `List` to store all vehicle instances. Implement a method to display vehicle details. Ensure invalid vehicle types throw exceptions.

25. **Exception Handling + Collections + OOP**
    Implement a student management module where student details (ID, name, marks) are stored in an `ArrayList`. Provide a method to calculate the average marks. Handle exceptions for division by zero or missing data.
26. **MVC Pattern + Collections + OOP**
    Create a small component for a library management system using the MVC pattern. The model should store book data using a `List`, the view should display books, and the controller should handle operations like adding or removing books. Handle exceptions for invalid book IDs.
27. **Generics + OOP + Exception Handling**
    Design a generic `Calculator<T>` class supporting basic operations (`add`, `subtract`, `multiply`, `divide`). Ensure the class handles exceptions like division by zero and supports both integer and floating-point operations.
28. **Decorator Pattern + Collections + OOP**
    Implement a text processing module where the base class reads input text. Apply decorators to enhance functionalities like converting text to uppercase, removing spaces, or counting words. Use a `List<String>` to store processed versions.
29. **State Machine + Collections + Exception Handling**
    Develop a state management feature for a package delivery system. Use an `Enum` to track states (`PICKED_UP`, `IN_TRANSIT`, `DELIVERED`). Store package details using a `HashMap` and handle invalid state transitions with exceptions.
30. **Collections + Exception Handling + OOP**
    Create a customer rewards system using a `TreeMap` to store customer IDs and their reward points. Implement a method to update points and print the top 3 customers with the highest points. Handle exceptions for invalid IDs or insufficient data.

Threading

1. What is the difference between `Thread` class and `Runnable` interface?
2. Explain why `sleep()` and `wait()` are different in Java.
3. What happens if `start()` is called twice on the same thread object?
4. How does `join()` work in multithreading?
5. Why is `volatile` used in Java multithreading?
6. Explain the difference between `synchronized` block and `synchronized` method.
7. What is a deadlock? Provide a brief scenario.
8. How does a `ReentrantLock` differ from a `synchronized` block?
9. Why should `ExecutorService` be preferred over creating new threads using `Thread`?
10. What is the significance of the `Fork/Join` framework in Java?

Exceptions

1. Explain the difference between checked and unchecked exceptions.
2. Why is it a bad practice to catch `Exception` directly?
3. Can we have multiple `catch` blocks for a single `try` block? Explain with an example.
4. What happens if `finally` block has a `return` statement?
5. How is `throw` different from `throws` in Java?
6. Explain the concept of exception chaining with an example.
7. Why is it generally not recommended to suppress exceptions using empty `catch` blocks?
8. Provide a use case where a `Custom Exception` might be required.
9. How can resources be safely closed using `try-with-resources`?
10. Explain the impact of using `RuntimeException` for application-level exceptions.

API Handling

1. What are the key methods of the `HttpURLConnection` class in Java?
2. Explain how `RESTful API` requests can be handled using Java.
3. What is the role of `BufferedReader` in API response reading?
4. How can JSON responses be parsed using Java?
5. What exceptions should be handled when making HTTP requests in Java?
6. How can API response time be measured in Java?
7. Explain the concept of connection pooling in API handling.
8. What is the significance of status codes like `200`, `404`, and `500`?
9. How can authentication be handled in API requests using Java?
10. Describe how API rate limiting can be implemented in Java.

Collections

1. Explain the difference between `HashSet` and `TreeSet`.
2. Why is `LinkedHashMap` preferred when maintaining insertion order?
3. How is `ConcurrentHashMap` different from `HashMap`?
4. Explain the internal working of `ArrayList` when an element is added beyond its capacity.
5. What is the significance of `fail-fast` behavior in Java Collections?
6. Why are `Collections.synchronizedList()` and `CopyOnWriteArrayList` used in multithreading?
7. What is the purpose of `LinkedList` when compared to `ArrayList`?
8. Explain how `PriorityQueue` determines the priority of elements.
9. How is a `Deque` different from a `Queue`?
10. Provide a scenario where a `WeakHashMap` is beneficial.

Collections

1. What is the difference between `ArrayList` and `LinkedList` in terms of insertion, deletion, and access time?
2. Explain how `HashMap` handles collisions using chaining.
3. Why would you use a `TreeSet` over a `HashSet`?
4. How does `LinkedHashMap` maintain the order of elements?
5. What is the significance of the `Comparator` and `Comparable` interfaces?
6. Explain the difference between `Vector` and `ArrayList`.
7. How does `Collections.unmodifiableList()` work? Provide a use case.
8. When would you prefer `PriorityQueue` over `LinkedList`?
9. Explain the internal working of a `HashMap` during resizing.
10. Why is the `ConcurrentSkipListSet` used in concurrent programming?

Searching in Collections

1. How can you efficiently search for an element in a `HashSet`?
2. Which search algorithm is used in `Arrays.binarySearch()`? What are its limitations?
3. Explain why `Collections.binarySearch()` requires the collection to be sorted.
4. How would you find the index of an object in an `ArrayList` without using built-in functions?
5. Provide a scenario where a `TreeMap` would be preferred for searching over a `HashMap`.
6. Why is searching in `LinkedList` slower than in `ArrayList`?
7. How does `NavigableSet` facilitate efficient searching?
8. Explain how searching is implemented in `TreeSet`.
9. What are the advantages of using `ConcurrentSkipListMap` for searching in concurrent applications?
10. How can you perform a reverse search using `Collections.reverseOrder()`?

Sorting in Collections

1. How does `Collections.sort()` differ from `Arrays.sort()`?
2. Explain the concept of natural ordering and how `Comparable` is used for sorting.
3. Provide a use case where using `Comparator` is preferred over `Comparable`.
4. Why would you choose a `TreeSet` for maintaining a sorted collection of unique elements?
5. What sorting algorithm does `Arrays.sort()` use for primitive data types and objects?
6. Explain how `PriorityQueue` maintains its sorted order.
7. How can you sort a list of custom objects based on multiple fields using `Comparator`?
8. Describe how `Collections.shuffle()` affects sorting.
9. What is the time complexity of `Collections.sort()` for different scenarios?
10. How can you sort a `Map` by its values instead of its keys using Java?

Collections and Wrapper Classes

1. Why is `ArrayList` preferred over `LinkedList` for random access?
2. What is the difference between `HashMap` and `TreeMap`?
3. Explain why `Integer` objects are immutable in Java.
4. Why is `Collections.unmodifiableList()` used? Give one use case.
5. Explain the role of `Comparator` and `Comparable` in sorting collections.
6. Why does `Double.NaN == Double.NaN` return false?
7. Explain the internal working of `HashSet` in Java.
8. Why is it recommended to override `equals()` and `hashCode()` when using custom objects in a `HashMap`?
9. How is a `PriorityQueue` different from a `Queue`?
10. What is the difference between `Optional` and `null` in Java?

Private Constructor and Factory Pattern

1. Why would a class have a private constructor? Provide an example use case.
2. How can a private constructor be accessed using a Factory Pattern?
3. Explain the Singleton Design Pattern using a private constructor.
4. What is the purpose of the Factory Method Pattern?
5. Why is a Factory Pattern preferred over simple object creation using `new`?

MVC Design Pattern

1. Explain the role of the Controller in an MVC architecture.
2. How does the Model interact with the View in MVC?
3. Why is loose coupling achieved using MVC?
4. Provide one real-world example where MVC is useful.
5. How can user input be validated in an MVC-based application?

Iterator and Decorator Pattern

1. What is the advantage of using an `Iterator` over a simple `for-loop`?
2. Explain how the Decorator Pattern follows the Open/Closed Principle.
3. How does `Iterator.remove()` work?
4. Provide an example of using the Decorator Pattern in Java.
5. What is the difference between `ListIterator` and `Iterator`?

Strategy and Annotation

1. How does the Strategy Pattern promote flexibility in code design?
2. What problem is solved by using the Strategy Pattern?
3. Explain the use of `@Override` annotation in Java.
4. How is `@FunctionalInterface` used in Java?
5. What is the purpose of `RetentionPolicy.RUNTIME` in custom annotations?

Generics and UML Diagrams

1. Explain the concept of type erasure in Java Generics.
2. Why is `List<?>` used in Java?
3. What is the difference between `List<? extends Number>` and `List<? super Number>`?
4. Provide an example where using Generics prevents runtime errors.
5. Draw a simple UML Class Diagram representing a `Vehicle` class and its subclasses `Car` and `Bike`.

---

1. Explain how Java achieves abstraction using abstract classes and interfaces. Provide a real-world scenario where both would be required, and justify why.
2. Suppose a class has private fields and public getter and setter methods. Why can this still be a security risk? Suggest how immutability can mitigate this risk.
3. A developer uses method overriding but accidentally hides the parent class method instead. What are the conditions under which this can happen? Provide a scenario where it might go unnoticed.
4. Identify the issue in the following code and explain the output:

```
class A {
    static void display() {
        System.out.println("Class A");
    }
}

class B extends A {
    static void display() {
        System.out.println("Class B");
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new B();
        obj.display();
    }
}
```

5. Explain how Java prevents memory leaks in case of circular references using garbage collection. How does `WeakReference` help in such scenarios?
6. Differentiate between upcasting and downcasting. Provide an example where downcasting might fail at runtime, even when it seems logically correct.

---

1. Consider the following scenario:
    - A class `Vehicle` has a method `drive()`.
    - Classes `Car` and `Bike` extend `Vehicle`.
    - A method takes `Vehicle` as input and calls the `drive()` method.

   Explain how polymorphism is applied here. Modify this scenario using an interface instead of a superclass. Would there be any functional advantage? Justify.

2. A developer implements a Singleton class using double-checked locking. However, the implementation sometimes results in multiple instances. Identify what might be wrong. Provide a correct implementation using the synchronized block.
3. Explain how constructor chaining works in Java. Create a scenario where improper constructor chaining leads to unexpected behavior. Suggest how to resolve it.

---

1. Design a class structure using the concepts of inheritance and interfaces to model a **Library Management System**. The system must include:
   - `Book` and `Member` as entities.
   - `Librarian` with access to book management.
   - `Borrower` with book borrowing rights.
   - Provide appropriate methods and apply abstraction where necessary.

   Describe how the relationships between classes would be implemented using OOP principles.

2. Analyze the following scenario:
   - A class `Order` has a list of `Product` objects.
   - `Product` has properties like name, price, and category.
   - The class needs to calculate the total order price and apply discounts based on product categories.

   Design the class structure using aggregation or composition. Justify which one is more appropriate in this scenario. Explain how the classes would interact and how OOP principles are applied.

---

1. **Identify the Problem and Fix It:**
   What will be the output of the following code? Explain the underlying issue.

```
class Parent {
    public Parent() {
        System.out.println("Parent constructor called");
        printMessage();
    }

    public void printMessage() {
        System.out.println("Message from Parent");
    }
}

class Child extends Parent {
    private String message = "Message from Child";

    public Child() {
        System.out.println("Child constructor called");
    }

    @Override
    public void printMessage() {
        System.out.println(message);
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
    }
}
```

- Why does the program print `null` instead of `"Message from Child"`?
- How can you fix it without changing the method call inside the constructor?

---

2. **Identify Output and Justify:**
   What will the following code print?

```
class A {
    static void show() {
        System.out.println("A's show");
    }
}

class B extends A {
    static void show() {
        System.out.println("B's show");
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new B();
        obj.show();
    }
}
```

- Explain the concept behind the output.
- What would happen if `show()` were not `static`?

---

3. **Abstract Class Misuse:**
   What issue will occur in this code? Fix it.

```
abstract class Shape {
    abstract void draw();
    void display() {
        System.out.println("Displaying Shape");
    }
}

class Circle extends Shape {
    // No implementation of draw()
}

public class Test {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.display();
    }
}
```

- Why does this code fail at runtime?
- Implement a proper fix using best OOP practices.

4. **Constructor Misuse with Superclass:**
   Identify the problem in this code and suggest a fix.

```
class Animal {
    Animal(String name) {
        System.out.println("Animal: " + name);
    }
}

class Dog extends Animal {
    Dog() {
        System.out.println("Dog created");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
    }
}
```

- Why does the code produce an error?
- What concept is violated?
- Suggest a corrected version using appropriate constructor chaining.

5. **Inheritance and Polymorphism Confusion:**
   What will be the output of this code? Explain why.

```
class Vehicle {
    public void run() {
        System.out.println("Vehicle is running");
    }
}

class Car extends Vehicle {
    public void run() {
        System.out.println("Car is running");
    }
}

class Bike extends Vehicle {
    public void run() {
        System.out.println("Bike is running");
    }
}

public class Test {
    public static void main(String[] args) {
        Vehicle vehicle = new Car();
        vehicle.run();
        vehicle = new Bike();
        vehicle.run();
    }
}
```

- What concept is demonstrated here?
- How does polymorphism affect the output?

---

6. **Interface and Realization Misuse:**
   Identify the issue in this code and resolve it.

```
interface Device {
    void powerOn();
    void powerOff();
}

abstract class Mobile implements Device {
    // No implementation of methods
}

class Smartphone extends Mobile {
    public void powerOn() {
        System.out.println("Smartphone Powering On");
    }
}

public class Test {
    public static void main(String[] args) {
        Device phone = new Smartphone();
        phone.powerOn();
    }
}
```

- Why does this code produce an error?
- Provide the corrected implementation.

---

1. Identify the Output and Explain (Conceptual and Tricky)

```
public class ExceptionTest {
    public static void main(String[] args) {
        try {
            System.out.println("Start");
            int a = 10 / 0;
            System.out.println("After Division");
        } catch (ArithmeticException e) {
            System.out.println("Catch Block: " + e.getMessage());
        } finally {
            System.out.println("Finally Block Executed");
        }
        System.out.println("End of Program");
    }
}
```

- **Question:** Predict the output and explain the order of execution. What happens if the exception wasn't an `ArithmeticException`?

---

2. Nested Try-Catch and Resource Management

```
import java.io.*;

public class NestedTryCatchExample {
    public static void main(String[] args) {
        FileReader fr = null;
        try {
            try {
                fr = new FileReader("nonexistentfile.txt");
                char[] a = new char[50];
                fr.read(a);
                System.out.println(a);
            } catch (IOException e) {
                System.out.println("Inner Catch Block: " + e);
                throw new RuntimeException("Exception in Outer Try");
            } finally {
                System.out.println("Inner Finally Executed");
            }
        } catch (RuntimeException e) {
            System.out.println("Outer Catch Block: " + e.getMessage());
        } finally {
            System.out.println("Outer Finally Executed");
            try {
                if (fr != null) {
                    fr.close();
                }
            } catch (IOException e) {
                System.out.println("Error Closing File: " + e);
            }
        }
    }
}
```

- **Question:** Predict the output. Explain how exceptions are handled in nested try-catch blocks. What happens if the file exists?

---

3. Custom Exception and Debugging

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class VoterEligibility {
    public static void checkEligibility(int age) throws InvalidAgeException
{
        if (age < 18) {
            throw new InvalidAgeException("Age is below 18");
        } else {
            System.out.println("Eligible to vote");
        }
    }

    public static void main(String[] args) {
        try {
            checkEligibility(16);
        } catch (InvalidAgeException e) {
            System.out.println("Exception Caught: " + e.getMessage());
        }
    }
}
```

- **Question:** Modify this code to catch multiple exceptions, including `InvalidAgeException` and `NumberFormatException`.
- **Follow-Up:** Explain the significance of custom exceptions in large-scale applications.

---

4. Exception Propagation and Debugging

```
public class ExceptionPropagation {
    public void methodA() {
        methodB();
    }

    public void methodB() {
        methodC();
    }

    public void methodC() {
        throw new ArithmeticException("Exception in Method C");
    }

    public static void main(String[] args) {
        ExceptionPropagation obj = new ExceptionPropagation();
        try {
            obj.methodA();
        } catch (Exception e) {
            System.out.println("Caught in Main: " + e.getMessage());
        }
    }
}
```

- **Question:** Explain how exception propagation works in this scenario. What will happen if we remove the try-catch block from `main()`?

5. Misleading Exception and Debugging Challenge

```
public class MisleadingException {
    public static void main(String[] args) {
        try {
            Object obj = Integer.valueOf(100);
            String str = (String) obj;
        } catch (ClassCastException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

- **Question:** Identify the issue and explain why a `ClassCastException` occurs. How can this be resolved?

1. Deadlock Identification

```
public class DeadlockExample {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void method1() {
        synchronized (lock1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            synchronized (lock2) {
                System.out.println("Thread 1: Acquired lock 2.");
            }
        }
    }

    public void method2() {
        synchronized (lock2) {
            System.out.println("Thread 2: Holding lock 2...");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            synchronized (lock1) {
                System.out.println("Thread 2: Acquired lock 1.");
            }
        }
    }

    public static void main(String[] args) {
        DeadlockExample example = new DeadlockExample();
        Thread t1 = new Thread(() -> example.method1());
        Thread t2 = new Thread(() -> example.method2());

        t1.start();
        t2.start();
    }
}
```

- **Question:** Analyze and explain why a deadlock might occur in this code. Provide suggestions on how to prevent it.

---

2. Race Condition and Debugging

```
public class RaceConditionExample implements Runnable {
    private int counter = 0;

    public void increment() {
        counter++;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            increment();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        RaceConditionExample example = new RaceConditionExample();
        Thread t1 = new Thread(example);
        Thread t2 = new Thread(example);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Counter Value: " + example.counter);
    }
}
```

- **Question:** Explain the concept of race conditions in this example. How can synchronization resolve the issue?

---

3. Thread Interruption and Handling

```
public class InterruptExample extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread running: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        InterruptExample thread = new InterruptExample();
        thread.start();
        try {
            Thread.sleep(2000);
            thread.interrupt();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- **Question:** Explain how `Thread.interrupt()` works. What will be the output if the interruption occurs during sleep?

---

4. Thread Priority and Misconceptions

```
public class ThreadPriorityExample extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Priority: "
+ Thread.currentThread().getPriority());
    }

    public static void main(String[] args) {
        ThreadPriorityExample t1 = new ThreadPriorityExample();
        ThreadPriorityExample t2 = new ThreadPriorityExample();

        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
    }
}
```

- **Question:** Will the thread with higher priority always execute first? Explain the role of thread priorities in Java.

---

5. Callable and Future Conceptual Question

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

public class CallableExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Callable<Integer> task = () -> {
            System.out.println("Task Executed");
            return 42;
        };

        Future<Integer> result = executor.submit(task);

        System.out.println("Result: " + result.get());
        executor.shutdown();
    }
}
```

- **Question:** Explain how `Callable` and `Future` work together. What are the advantages of using `Callable` over `Runnable`?

---

1. Issue with HashMap Key Behavior

```
import java.util.HashMap;
import java.util.Map;

class Key {
    int id;

    Key(int id) {
        this.id = id;
    }

    @Override
    public boolean equals(Object obj) {
        return true; // Always returns true
    }
}

public class Main {
    public static void main(String[] args) {
        Map<Key, String> map = new HashMap<>();
        map.put(new Key(1), "One");
        map.put(new Key(2), "Two");
        map.put(new Key(3), "Three");

        System.out.println("Map Size: " + map.size());
    }
}
```

- **Question:**
  - What will be the output? Why?
  - How can we fix this issue?

Answer:

- The output will be `Map Size: 3`.
- Even though `equals()` always returns `true`, the map uses both `hashCode()` and `equals()` for comparisons. Since no `hashCode()` is overridden, the default hashCode from `Object` is used, resulting in different hash buckets.
- Fix: Implement both `equals()` and `hashCode()` methods properly.

---

2. Problem with ConcurrentModificationException

```
import java.util.*;

public class ConcurrentExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("One", "Two",
"Three", "Four"));
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String str = iterator.next();
            if (str.equals("Three")) {
                list.remove(str); // Throws Exception
            }
        }
    }
}
```

- **Question:**
    - Why does this code throw a `ConcurrentModificationException`?
    - How can it be fixed?

Answer:

- The code throws a `ConcurrentModificationException` because removing elements directly from the list while using an iterator is unsafe.
- **Fix 1:** Use `iterator.remove()` instead of `list.remove()`.
- **Fix 2:** Use `ListIterator` or `CopyOnWriteArrayList` for modification during iteration.

---

3. Misunderstanding PriorityQueue

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.add(40);
        pq.add(10);
        pq.add(30);
        pq.add(20);

        System.out.println(pq.peek()); // ?
    }
}
```

- **Question:**
    - What will be the output?
    - Explain why the result may not be what a student might expect.

Answer:

- The output will be `10` because a `PriorityQueue` is a **Min-Heap** by default, meaning it keeps the smallest element at the front.
- Some students may expect the order to follow insertion, but `PriorityQueue` is not a FIFO queue.

---

4. Unpredictable HashSet Behavior

```
import java.util.*;

public class HashSetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Mango");
        set.add("Banana");
        System.out.println(set);
    }
}
```

- **Question:**
    - Why does "Banana" not appear twice in the output?
    - What is the time complexity of adding elements to a HashSet?

Answer:

- HashSet does not allow duplicates. When "Banana" is added a second time, it checks using equals() and hashCode() and rejects the duplicate.
- The time complexity for adding elements is **O(1)** on average, but it can be **O(n)** in the worst case due to hash collisions.

---

5. Understanding Collections.sort() and Comparable

```
import java.util.*;

class Student implements Comparable<Student> {
    String name;
    int marks;

    Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    @Override
    public int compareTo(Student other) {
        return Integer.compare(this.marks, other.marks);
    }
}

public class SortExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("John", 85));
        students.add(new Student("Alice", 95));
        students.add(new Student("Bob", 75));

        Collections.sort(students);

        for (Student s : students) {
            System.out.println(s.name + " - " + s.marks);
        }
    }
}
```

- **Question:**
    - How will the students be sorted?
    - How can we sort the students in descending order?

Answer:

- The students will be sorted in **ascending order of marks** because compareTo() uses Integer.compare().

- To sort in descending order, use `Collections.sort(students, Comparator.reverseOrder())` or change the comparison logic to `return Integer.compare(other.marks, this.marks);`.

---

6. Improper Use of TreeSet

```
import java.util.*;

public class TreeSetExample {
    public static void main(String[] args) {
        Set<String> treeSet = new TreeSet<>();
        treeSet.add("Apple");
        treeSet.add(null);
        treeSet.add("Banana");
        System.out.println(treeSet);
    }
}
```

- **Question:**
  - What error will occur? Explain why.

Answer:

- A `NullPointerException` will be thrown because `TreeSet` uses natural ordering (via `Comparable` or `Comparator`) and does not support `null` values.
- Unlike `HashSet`, which allows one `null` value, `TreeSet` relies on comparison to maintain order, making `null` comparisons impossible.

---

7. Misunderstanding LinkedHashMap

```
import java.util.*;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new LinkedHashMap<>(3, 0.75f, true);
        map.put("A", 1);
        map.put("B", 2);
        map.put("C", 3);

        map.get("A");
        map.get("B");

        System.out.println(map);
    }
}
```

- **Question:**
  - What will be the order of elements in the map?
  - What specific feature of `LinkedHashMap` is demonstrated here?

Answer:

- The output will be `{C=3, A=1, B=2}`.

- `LinkedHashMap` with the third parameter set to `true` uses **access-order** instead of insertion-order.
- Since `"A"` and `"B"` were accessed, they moved to the end of the order, demonstrating its **LRU (Least Recently Used) Cache behavior**.

---

1. Misinterpretation of HashMap with Null Key and Values

```
import java.util.*;

public class HashMapExample {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put(null, "First");
        map.put("Key", null);
        map.put(null, "Second");
        System.out.println(map);
    }
}
```

- **Question:**
  - What will be the output? Explain why.

Answer:

- The output will be `{null=Second, Key=null}`.
- `HashMap` allows one `null` key and multiple `null` values.
- The second `put()` with `null` key overwrites the first entry because keys must be unique.

---

2. Incorrect Use of Comparable in TreeSet

```
import java.util.*;

class Student implements Comparable<Student> {
    String name;
    int age;

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Student other) {
        return this.name.compareTo(other.name);
    }
}

public class TreeSetExample {
    public static void main(String[] args) {
        Set<Student> set = new TreeSet<>();
        set.add(new Student("Alice", 22));
        set.add(new Student("Bob", 25));
        set.add(new Student("Alice", 23));

        System.out.println(set.size());
    }
}
```

- **Question:**
  - What will be the output? Explain the reasoning.

Answer:

- The output will be `2`.
- `TreeSet` relies on the `compareTo()` method for ordering.
- Since the comparison is done only on `name`, it treats both `"Alice"` objects as duplicates.
- To fix this, modify `compareTo()` to consider both `name` and `age`.

---

3. LinkedList Memory Management

```
import java.util.*;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        list.add(10);
        list.add(20);
        list.add(30);
        list.remove(1);
        list.addFirst(5);
        list.addLast(40);

        System.out.println(list);
    }
}
```

- **Question:**
    - What will be the output?
    - Explain the behavior of `addFirst()` and `addLast()`.

Answer:

- Output: `[5, 10, 30, 40]`
- `addFirst(5)` inserts `5` at the head.
- `addLast(40)` appends `40` to the tail.
- `remove(1)` removes the element at index `1` (which was `20`).

---

4. Understanding PriorityQueue with Custom Comparator

```
import java.util.*;

public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<String> pq = new
PriorityQueue<>(Comparator.reverseOrder());
        pq.add("Banana");
        pq.add("Apple");
        pq.add("Mango");

        while (!pq.isEmpty()) {
            System.out.println(pq.poll());
        }
    }
}
```

- **Question:**
    - What will be the output?

    - Why is the output reversed?

Answer:

- Output:
- `Mango`
- `Banana`
- `Apple`
- `PriorityQueue` with `Comparator.reverseOrder()` turns it into a **Max-Heap**.
- Elements are removed in descending order, starting from the largest lexicographical value.

---

5. Unexpected Behavior with Collections.unmodifiableList

```
import java.util.*;

public class UnmodifiableListExample {
    public static void main(String[] args) {
        List<String> originalList = new ArrayList<>(Arrays.asList("A", "B",
"C"));
        List<String> unmodifiableList =
Collections.unmodifiableList(originalList);

        originalList.add("D");
        System.out.println(unmodifiableList);
    }
}
```

- **Question:**
    - Will the unmodifiable list reflect the change? Why or why not?

Answer:

- Yes, the output will be `[A, B, C, D]`.
- `Collections.unmodifiableList()` creates a **read-only view** of the original list.
- Since the original list is modified, the view reflects those changes.
- However, any attempt to modify the `unmodifiableList` directly would throw `UnsupportedOperationException`.

---

6. Improper Usage of Queue with Null Values

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new PriorityQueue<>();
        queue.add("John");
        queue.add(null);
        queue.add("Jane");
        System.out.println(queue);
    }
}
```

- **Question:**

- What will happen when this code is executed?

Answer:

- It will throw a `NullPointerException`.
- `PriorityQueue` does not allow `null` values because it requires elements to be comparable for ordering.
- Use a `LinkedList` instead if null values are required.

---

7. Stack vs Deque Misconception

```
import java.util.*;

public class DequeExample {
    public static void main(String[] args) {
        Deque<Integer> deque = new ArrayDeque<>();
        deque.push(10);
        deque.push(20);
        deque.push(30);

        System.out.println(deque.pop());
        System.out.println(deque.remove());
    }
}
```

- **Question:**
  - What will be the output?
  - How is the behavior of `Deque` different from `Stack` here?

Answer:

- Output:
- `30`
- `20`
- `Deque` acts as a double-ended queue, but `push()` and `pop()` mimic `Stack` behavior (LIFO).
- `remove()` here acts like a queue and removes from the front.

---

8. Wrong Use of Comparator with TreeMap

```
import java.util.*;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> map = new TreeMap<>((a, b) -> 0);
        map.put("A", 10);
        map.put("B", 20);
        map.put("C", 30);
        System.out.println(map);
    }
}
```

- **Question:**
  - What will be the output? Explain the mistake.

Answer:

- The output will be `{A=10}`.
- The comparator `(a, b) -> 0` treats all keys as equal, meaning every subsequent `put()` overwrites the previous entry.
- Fix: Use a proper comparator that differentiates keys, such as `Comparator.naturalOrder()`.

---

1. Static Method Overriding Confusion

```
class Parent {
    public static void display() {
        System.out.println("Parent Display");
    }
}

class Child extends Parent {
    public static void display() {
        System.out.println("Child Display");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        Parent obj2 = new Child();
        obj1.display();
        obj2.display();
    }
}
```

- **Question:**
  - What will be the output? Explain.

Answer:

- Output:
- `Parent Display`
- `Parent Display`
- Static methods are **not overridden** but **hidden**.
- The method call is resolved at **compile-time** using reference type (`Parent`).
- Therefore, both calls execute the `Parent` class's `display()` method.

---

2. Ambiguous Overloading with Autoboxing

```
class Test {
    public void method(int i) {
        System.out.println("int method");
    }

    public void method(Integer i) {
        System.out.println("Integer method");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.method(10);
    }
}
```

- **Question:**
  - Which method will be called and why?

Answer:

- Output: `int method`
- **Primitive types** take precedence over their wrapper classes during method selection.
- `int` is a more specific match compared to `Integer`, so the `int method` is called.

---

3. Final Method in Polymorphism

```
class Parent {
    public final void display() {
        System.out.println("Parent Display");
    }
}

class Child extends Parent {
    public void display() {
        System.out.println("Child Display");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.display();
    }
}
```

- **Question:**
    - What will happen when this code is executed?

Answer:

- Compilation Error.
- `final` methods cannot be overridden.
- The compiler detects the invalid attempt to override the `display()` method in `Child`.

---

4. Private Method Overriding Misconception

```
class Parent {
    private void display() {
        System.out.println("Parent Display");
    }
}

class Child extends Parent {
    public void display() {
        System.out.println("Child Display");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.display();
    }
}
```

- **Question:**
    - What will be the output? Explain the behavior.

Answer:

- Compilation Error.
- Private methods are **not inherited** and cannot be overridden.
- The call `p.display()` will fail because `display()` in `Parent` is private and inaccessible from `Child`.

---

5. Constructor Chaining with Inheritance

```
class A {
    A() {
        System.out.println("A's Constructor");
    }
}

class B extends A {
    B() {
        System.out.println("B's Constructor");
    }
}

class C extends B {
    C() {
        System.out.println("C's Constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        C obj = new C();
    }
}
```

- **Question:**
    - What will be the output? Explain the constructor chaining mechanism.

Answer:

- Output:
- `A's Constructor`
- `B's Constructor`
- `C's Constructor`
- Constructor chaining follows a **top-down** approach.
- The constructor of the parent class (`A`) is called first, followed by `B`, and then `C`.

---

6. Covariant Return Type with Method Overriding

```
class Parent {
    public Number getValue() {
        return 10;
```

```java
    }
}

class Child extends Parent {
    @Override
    public Integer getValue() {
        return 20;
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        System.out.println(p.getValue());
    }
}
```

- **Question:**
  - What will be the output? Explain why covariant return types work.

Answer:

- Output: `20`
- Covariant return types allow the overriding method to return a **subtype** (`Integer`) of the parent's return type (`Number`).
- Polymorphism ensures the `Child` class's `getValue()` method is called at runtime.

---

7. Polymorphism with Overloading

```java
class Parent {
    public void show(String s) {
        System.out.println("Parent String");
    }

    public void show(Object o) {
        System.out.println("Parent Object");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.show(null);
    }
}
```

- **Question:**
  - What will be the output?

Answer:

- Output: `Parent String`
- Since `null` can match both `String` and `Object`, Java prefers the most **specific match** (`String`).

8. Use of `super()` with Parameterized Constructor

```
class Parent {
    Parent(String message) {
        System.out.println("Parent: " + message);
    }
}

class Child extends Parent {
    Child() {
        super("Hello from Parent");
        System.out.println("Child Constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        new Child();
    }
}
```

- **Question:**
  - What will be the output?

Answer:

- Output:
- `Parent: Hello from Parent`
- `Child Constructor`
- `super()` is used to invoke the parent's constructor with a parameter.
- After `Parent` completes execution, the `Child` constructor runs.

9. Static Block and Instance Block Execution

```
class Parent {
    static {
        System.out.println("Parent Static Block");
    }
    {
        System.out.println("Parent Instance Block");
    }
    Parent() {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent {
    static {
        System.out.println("Child Static Block");
    }
    {
        System.out.println("Child Instance Block");
    }
    Child() {
        System.out.println("Child Constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

- **Question:**
  - What will be the output? Explain the order of execution.

Answer:

- Output:
- `Parent Static Block`
- `Child Static Block`
- `Parent Instance Block`
- `Parent Constructor`
- `Child Instance Block`
- `Child Constructor`
- **Explanation:**
  - Static blocks execute **once** during class loading.
  - Parent's static block runs first, then Child's static block.
  - During object creation, instance blocks and constructors execute in the order of inheritance (Parent first, then Child).

---

10. Downcasting and ClassCastException

```
class Parent { }
class Child extends Parent { }

public class Test {
    public static void main(String[] args) {
        Parent p = new Parent();
        Child c = (Child) p;  // Downcasting
    }
}
```

- **Question:**
  - What will happen at runtime?

Answer:

- **Runtime Exception:** `ClassCastException`
- **Explanation:**
  - Downcasting requires the object to be of the type being cast.
  - `Parent p` is not actually a `Child`, so the cast will fail at runtime.

---

11. Method Hiding with Static Methods

```
class Parent {
    public static void print() {
        System.out.println("Parent Print");
    }
}

class Child extends Parent {
    public static void print() {
        System.out.println("Child Print");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.print();
    }
}
```

- **Question:**
    - What will be the output?

Answer:

- Output: `Parent Print`
- **Explanation:**
    - Static methods are **hidden** (not overridden) in inheritance.
    - Method binding is done at **compile time** based on reference type (`Parent`).

---

12. Method Overloading with Varargs

```
class Test {
    public void display(String s) {
        System.out.println("String method");
    }

    public void display(Object o) {
        System.out.println("Object method");
    }

    public void display(String... s) {
        System.out.println("Varargs method");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.display("Hello");
    }
}
```

- **Question:**
    - Which method will be invoked? Why?

Answer:

- Output: `String method`
- **Explanation:**
    - The exact match (`String`) is preferred over the `Object` method or the varargs method.
    - Varargs is the least preferred in method resolution.

---

13. Overriding with Covariant Return Types

```
class Parent {
    public Number getValue() {
        return 10;
    }
}

class Child extends Parent {
    @Override
    public Double getValue() {
        return 20.5;
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        System.out.println(p.getValue());
    }
}
```

- **Question:**
    - What will be the output?

Answer:

- Output: `20.5`
- **Explanation:**
    - Java supports **covariant return types** in overriding.
    - `Child` returns a `Double`, which is a subclass of `Number`.
    - Polymorphism ensures the correct method from `Child` is invoked.

---

14. Abstract Class with Final Method

```
abstract class AbstractParent {
    public final void display() {
        System.out.println("Final Display");
    }
}

class Child extends AbstractParent {
    public void display() {
        System.out.println("Child Display");
    }
}

public class Test {
    public static void main(String[] args) {
        AbstractParent p = new Child();
        p.display();
    }
}
```

- **Question:**
  - Will this code compile? If not, why?

Answer:

- **Compilation Error:**
  - `final` methods cannot be overridden.
  - Attempting to override `display()` in the `Child` class results in a compilation error.

---

15. Understanding `super()` and Constructor Chaining

```
class A {
    A() {
        System.out.println("Class A Constructor");
    }
}

class B extends A {
    B() {
        super();
        System.out.println("Class B Constructor");
    }
}

class C extends B {
    C() {
        System.out.println("Class C Constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        new C();
    }
}
```

- **Question:**
  - What will be the output?

Answer:

- Output:
- `Class A Constructor`
- `Class B Constructor`
- `Class C Constructor`
- **Explanation:**
  - Every constructor in Java implicitly calls `super()` unless specified.
  - The constructor chain starts from the top of the inheritance hierarchy.

16. Multiple Inheritance Using Interfaces

```
interface A {
    default void display() {
        System.out.println("Display from A");
    }
}

interface B {
    default void display() {
        System.out.println("Display from B");
    }
}

class C implements A, B {
    @Override
    public void display() {
        A.super.display();
        B.super.display();
        System.out.println("Display from C");
    }
}

public class Test {
    public static void main(String[] args) {
        C obj = new C();
        obj.display();
    }
}
```

- **Question:**
    - What will be the output? Explain how conflicts are resolved.

Answer:

- Output:
- `Display from A`
- `Display from B`
- `Display from C`
- **Explanation:**
    - Java uses the `A.super.display()` and `B.super.display()` to resolve multiple inheritance conflicts using interfaces.
    - Without these calls, a compile-time error would occur.

17. Abstract Class with Static Method

```
abstract class Parent {
    public static void display() {
        System.out.println("Static Display in Parent");
    }
}

class Child extends Parent {
    public static void display() {
        System.out.println("Static Display in Child");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent.display();
        Child.display();
    }
}
```

- **Question:**
    - What will be the output? Why?

Answer:

- Output:
- `Static Display in Parent`
- `Static Display in Child`
- **Explanation:**
    - **Static methods** belong to the class, not the instance.
    - They are not overridden but **hidden**.
    - Calling `Parent.display()` executes the parent's version.

---

18. Private Method Overriding

```
class Parent {
    private void show() {
        System.out.println("Parent Show");
    }
}

class Child extends Parent {
    public void show() {
        System.out.println("Child Show");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show();
    }
}
```

- **Question:**
    - Will this compile? If not, why?

Answer:

- **Compilation Error:**
    - The method `show()` in the parent is **private** and not inherited by the child.
    - Even though the child has a public method with the same name, it is **not overriding** but simply a separate method.
    - **Solution:** Remove `private` or use a different method signature.

---

19. Superclass Reference and Method Overriding

```
class Parent {
    public void show() {
        System.out.println("Parent Show");
    }
}

class Child extends Parent {
    public void show() {
        System.out.println("Child Show");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show();
    }
}
```

- **Question:**
    - What will be the output? Explain why.

Answer:

- Output:
- `Child Show`
- **Explanation:**
    - Method overriding in Java uses **dynamic method dispatch** (runtime polymorphism).
    - Even though the reference is of type `Parent`, the actual object is of type `Child`, so the child's method is called.

---

20. Constructor and Method Overloading Ambiguity

```
class Test {
    Test(int a, long b) {
        System.out.println("int and long");
    }

    Test(long a, int b) {
        System.out.println("long and int");
    }

    public static void main(String[] args) {
        new Test(10, 20); // Ambiguous call
    }
}
```

- **Question:**
    - Will this code compile? If not, why?

Answer:

- **Compilation Error:**
    - Both constructors are equally valid for `new Test(10, 20)` since both `int` and `long` are possible matches.
    - This creates **ambiguity** at compile time.
    - **Solution:** Use explicit type casting like `new Test(10L, 20)` to resolve ambiguity.

---

21. Static Variable Shadowing

```
class Parent {
    static int value = 10;
}

class Child extends Parent {
    static int value = 20;
}

public class Test {
    public static void main(String[] args) {
        System.out.println(Parent.value);
        System.out.println(Child.value);
        Parent p = new Child();
        System.out.println(p.value);
    }
}
```

- **Question:**
    - What will be the output? Explain variable shadowing.

Answer:

- Output:
- `10`
- `20`
- `10`
- **Explanation:**
    - **Static variables** are class-level and cannot be overridden.
    - `Parent p = new Child();` uses the reference type to access the static variable.
    - Therefore, `p.value` will print `10` from the `Parent` class.

---

22. Overloading with Wrapper Classes

```
public class Test {
    public void print(Integer i) {
        System.out.println("Integer Method");
    }

    public void print(int i) {
        System.out.println("int Method");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.print(5);
        Integer obj = 5;
        t.print(obj);
    }
}
```

- **Question:**
  - What will be the output?

Answer:

- Output:
- `int Method`
- `Integer Method`
- **Explanation:**
  - For `t.print(5)` the primitive `int` version is called.
  - For `t.print(obj)`, since `obj` is an `Integer`, the wrapper class method is used.

---

23. Final Method and Inheritance

```
class Parent {
    public final void display() {
        System.out.println("Final Display in Parent");
    }
}

class Child extends Parent {
    public void display() {
        System.out.println("Display in Child");
    }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
    }
}
```

- **Question:**
  - Will this code compile? If not, why?

Answer:

- **Compilation Error:**
  - The `display()` method in `Parent` is **final** and cannot be overridden.
  - Final methods are designed to prevent further overriding in subclasses.

---

1. Abstract Class with Final Method

```
abstract class A {
    final void display() {
        System.out.println("Final method in Abstract Class A");
    }
    abstract void show();
}

class B extends A {
    void show() {
        System.out.println("Show in Class B");
    }
}

public class Test {
    public static void main(String[] args) {
        B b = new B();
        b.display();
        b.show();
    }
}
```

- **Question:**
    - Will this code run successfully? Why or why not?

Answer:

- **Yes.**
- Output:
- `Final method in Abstract Class A`
- `Show in Class B`
- **Explanation:**
    - Abstract classes can have **final methods**.
    - Final methods cannot be overridden in subclasses.

---

2. Multiple Inheritance with Abstract Classes

```
abstract class X {
    abstract void methodX();
}

abstract class Y {
    abstract void methodY();
}

class Z extends X {
    void methodX() {
        System.out.println("Method X in Class Z");
    }
}

public class Test {
    public static void main(String[] args) {
        Z obj = new Z();
        obj.methodX();
    }
}
```

- **Question:**
    - Will this code compile? Why or why not?

Answer:

- **Yes.**
- Output:
- `Method X in Class Z`
- **Explanation:**
    - **Java** does not support multiple inheritance with classes, but multiple interfaces are allowed.
    - Only one class can be extended at a time.

---

3. Interface Extending Multiple Interfaces

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

interface C extends A, B {
    void methodC();
}

class D implements C {
    public void methodA() {
        System.out.println("Method A in Class D");
    }

    public void methodB() {
        System.out.println("Method B in Class D");
    }

    public void methodC() {
        System.out.println("Method C in Class D");
    }
}

public class Test {
    public static void main(String[] args) {
        D d = new D();
        d.methodA();
        d.methodB();
        d.methodC();
    }
}
```

- **Question:**
    - Explain how multiple inheritance works with interfaces in Java.

Answer:

- Output:
- `Method A in Class D`
- `Method B in Class D`
- `Method C in Class D`
- **Explanation:**
  - Interfaces support **multiple inheritance** using the `extends` keyword.
  - The implementing class **D** provides definitions for all inherited methods.

---

4. Final Class with Static Methods

```
final class FinalClass {
    static void printMessage() {
        System.out.println("Hello from FinalClass");
    }
}

public class Test {
    public static void main(String[] args) {
        FinalClass.printMessage();
    }
}
```

- **Question:**
  - Can a final class have static methods? Explain.

Answer:

- **Yes.**
- Output:
- `Hello from FinalClass`
- **Explanation:**
  - **Final classes** cannot be subclassed, but they can contain static methods.
  - Static methods are accessed using the class name.

---

5. Static Methods in Interfaces

```
interface InterfaceA {
    static void staticMethod() {
        System.out.println("Static Method in InterfaceA");
    }
}

public class Test {
    public static void main(String[] args) {
        InterfaceA.staticMethod();
    }
}
```

- **Question:**
  - Explain why static methods are allowed in interfaces.

Answer:

- Output:
- `Static Method in InterfaceA`
- **Explanation:**
  - **Static methods** in interfaces are used for utility or helper methods.
  - They are not inherited by implementing classes and can only be accessed using the interface name.

---

6. Anonymous Inner Class with Interface

```
interface Greeting {
    void sayHello();
}

public class Test {
    public static void main(String[] args) {
        Greeting greeting = new Greeting() {
            public void sayHello() {
                System.out.println("Hello from Anonymous Class");
            }
        };
        greeting.sayHello();
    }
}
```

- **Question:**
  - What will be the output?

Answer:

- Output:
- `Hello from Anonymous Class`
- **Explanation:**
  - The interface is implemented using an **anonymous inner class**.
  - It provides the method implementation on the spot.

---

7. Private Inner Class with Access Method

```
class OuterClass {
    private class InnerClass {
        void show() {
            System.out.println("Show from Inner Class");
        }
    }

    void accessInnerClass() {
        InnerClass inner = new InnerClass();
        inner.show();
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.accessInnerClass();
    }
}
```

- **Question:**
    - Will the private inner class be accessible?

Answer:

- **Yes.**
- Output:
- `Show from Inner Class`
- **Explanation:**
    - A private inner class is accessible within its outer class.
    - The `accessInnerClass()` method acts as a gateway to access it.

---

8. Error Due to Private Constructor

```
class Singleton {
    private static Singleton instance;
    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

public class Test {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();
        System.out.println(obj1 == obj2);
    }
}
```

- **Question:**
    - What is the purpose of the private constructor?

Answer:

- Output:
- `true`
- **Explanation:**
    - The private constructor prevents direct instantiation.
    - The class ensures a **single instance** using the `getInstance()` method, implementing the **Singleton Design Pattern**.

1. Multiple Inheritance and Static Block Confusion

```
interface A {
    default void display() {
        System.out.println("Display from A");
    }
}

interface B {
    default void display() {
        System.out.println("Display from B");
    }
}

class C implements A, B {
    static {
        System.out.println("Static Block in C");
    }

    public void display() {
        System.out.println("Display from Class C");
        A.super.display();
    }
}

public class Test {
    public static void main(String[] args) {
        C c = new C();
        c.display();
    }
}
```

Question:

- What will be the output of the code?
- Explain how the method resolution works.

Answer:

- **Output:**
- `Static Block in C`
- `Display from Class C`
- `Display from A`
- **Explanation:**
    - Static blocks execute **once** when the class is loaded.
    - The `A.super.display()` is used to explicitly call the default method from interface `A`.
    - Java requires explicit disambiguation when both interfaces have conflicting methods.

---

2. Exception Propagation with Multiple Catch Blocks

```
class A {
    void method() throws ArithmeticException {
        throw new ArithmeticException("Exception from A");
    }
}

class B extends A {
    void method() throws NullPointerException {
        throw new NullPointerException("Exception from B");
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new B();
        try {
            obj.method();
        } catch (NullPointerException e) {
            System.out.println("Caught: " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

Question:

- Will the code compile?
- Predict the output.

Answer:

- **Output:**
- `Caught: Exception from A`
- **Explanation:**
    - The reference is of type `A`, so its version of `method()` is called.
    - Since it throws an `ArithmeticException`, it is caught in the appropriate catch block.

---

3. Singleton with Factory Pattern and Exception Handling

```
class DatabaseConnection {
    private static DatabaseConnection instance;
    private DatabaseConnection() {
        if (instance != null) {
            throw new RuntimeException("Instance already exists!");
        }
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }
}

public class Test {
    public static void main(String[] args) {
        try {
            DatabaseConnection connection1 =
DatabaseConnection.getInstance();
            DatabaseConnection connection2 = new DatabaseConnection(); //
Attempting direct access
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Question:

- Why is the constructor throwing an exception?
- What is the purpose of this implementation?

Answer:

- **Output:**
- `Instance already exists!`
- **Explanation:**
  - The constructor is protected with a check to enforce the **Singleton Pattern**.
  - The exception prevents any attempt to create additional instances using reflection or other techniques.

---

4. Inner Class and Interface Conflict

```
interface Greeting {
    void sayHello();
}

class Outer {
    private String message = "Hello from Outer";

    class Inner implements Greeting {
        public void sayHello() {
            System.out.println(message);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer().new Inner();
        inner.sayHello();
    }
}
```

Question:

- Explain how the inner class `Inner` can access the private variable `message`.

Answer:

- **Output:**
- `Hello from Outer`

- **Explanation:**
    - Inner classes have **direct access** to all private members of their enclosing class.
    - This is an advantage of using inner classes for encapsulation purposes.

---

5. Abstract Class, Interface, and Static Block Combination

```
interface I {
    default void print() {
        System.out.println("Interface I print()");
    }
}

abstract class A implements I {
    static {
        System.out.println("Static Block in A");
    }

    abstract void display();
}

class B extends A {
    static {
        System.out.println("Static Block in B");
    }

    void display() {
        System.out.println("Display in Class B");
    }
}

public class Test {
    public static void main(String[] args) {
        A a = new B();
        a.display();
        a.print();
    }
}
```

Question:

- What is the order of execution in this code?

Answer:

- **Output:**
- `Static Block in A`
- `Static Block in B`
- `Display in Class B`
- `Interface I print()`
- **Explanation:**
    - **Static blocks** execute when the class is first loaded, in superclass to subclass order.
    - `a.display()` calls the overridden method from `B`.
    - `a.print()` uses the interface's default method.

---

1. Interface and Static Method Conflict

```
interface A {
    static void print() {
        System.out.println("Static print in Interface A");
    }
}

interface B {
    static void print() {
        System.out.println("Static print in Interface B");
    }
}

class C implements A, B {
    void display() {
        // Attempting to call static methods
        A.print();
        B.print();
    }
}

public class Test {
    public static void main(String[] args) {
        C obj = new C();
        obj.display();
    }
}
```

Question:

- Will the code compile? If yes, what will be the output?
- Explain how the conflict between interface static methods is resolved.

Answer:

- **Output:**
- `Static print in Interface A`
- `Static print in Interface B`
- **Explanation:**
  - Static methods in interfaces are **not inherited** by implementing classes.
  - They are accessed using the interface name (`A.print()` and `B.print()`).
  - There is no conflict since static methods are called directly from the interface.

---

2. Abstract Class, Overloading, and Method Resolution

```
abstract class Parent {
    abstract void show(String s);
}

class Child extends Parent {
    void show(String s) {
        System.out.println("Child: " + s);
    }

    void show(Integer i) {
        System.out.println("Child (Integer): " + i);
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show("Hello");
        // p.show(10); // Compilation Error
    }
}
```

Question:

- Why does the second call (`p.show(10)`) cause a compilation error?
- How can this issue be resolved?

Answer:

- **Error:**
- `Compilation Error: Cannot resolve method 'show(int)'`
- **Explanation:**
  - The reference `p` is of type `Parent`, and `Parent` has no method `show(Integer)`.
  - **Overloading** is resolved at **compile time** based on reference type.
  - To fix it, cast to `Child`:
  - `((Child) p).show(10);`

---

3. Exception Handling with Inheritance and Method Overriding

```
class SuperClass {
    void process() throws Exception {
        System.out.println("Processing in SuperClass");
    }
}

class SubClass extends SuperClass {
    void process() throws RuntimeException {
        System.out.println("Processing in SubClass");
    }
}

public class Test {
    public static void main(String[] args) {
        SuperClass obj = new SubClass();
        try {
            obj.process();
        } catch (Exception e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

Question:

- Will the code compile? If yes, what will be the output?
- Explain how exception handling works with method overriding.

Answer:

- **Output:**
- `Processing in SubClass`
- **Explanation:**
    - In **method overriding**, a subclass can declare fewer or no exceptions than the superclass.
    - Since `SubClass` uses a **RuntimeException** (unchecked exception), the code works without any issues.

---

4. Abstract Class, Interface, and Constructor Confusion

```
interface A {
    void display();
}

abstract class B implements A {
    B() {
        System.out.println("Abstract Class B Constructor");
    }
}

class C extends B {
    C() {
        System.out.println("Class C Constructor");
    }

    public void display() {
        System.out.println("Display from Class C");
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new C();
        obj.display();
    }
}
```

Question:

- What will be the output?
- Explain the constructor call sequence.

Answer:

- **Output:**
- `Abstract Class B Constructor`
- `Class C Constructor`
- `Display from Class C`
- **Explanation:**
    - Even though the reference is of type `A`, the constructor chaining works from the abstract class to the subclass.
    - **Constructors** are always executed in the hierarchy from **superclass to subclass**.

    - Then, the overridden `display()` method in `C` is called.

---

5. Static, Final, and Object Manipulation

```
class Test {
    static int count = 0;

    static final Test instance = new Test();

    private Test() {
        count++;
    }

    static Test getInstance() {
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Test t1 = Test.getInstance();
        Test t2 = Test.getInstance();
        System.out.println("Count: " + Test.count);
    }
}
```

Question:

- What will be the output?
- Explain how `static final` impacts object creation.

Answer:

- **Output:**
- `Count: 1`
- **Explanation:**
    - `static final` ensures the instance is created **only once** using the **Singleton pattern**.
    - Even though `getInstance()` is called twice, both `t1` and `t2` refer to the same instance.
    - `count` is incremented only once during the first instantiation.

---

1. Multiple Interface Inheritance with Default Methods

```
interface A {
    default void display() {
        System.out.println("Display from A");
    }
}

interface B {
    default void display() {
        System.out.println("Display from B");
    }
}

class C implements A, B {
    @Override
    public void display() {
        A.super.display();
        B.super.display();
        System.out.println("Display from C");
    }
}

public class Test {
    public static void main(String[] args) {
        C obj = new C();
        obj.display();
    }
}
```

Question:

- What will be the output?
- Explain how Java resolves multiple default methods.

Answer:

- **Output:**
- `Display from A`
- `Display from B`
- `Display from C`
- **Explanation:**
    - When two interfaces provide the same default method, Java requires the implementing class to **resolve the conflict explicitly** using `InterfaceName.super.method()`.
    - Here, both `A.super.display()` and `B.super.display()` are called, followed by `C.display()`.

---

2. Static Block Execution Order

```
class Parent {
    static {
        System.out.println("Parent Static Block");
    }
}
```

```
class Child extends Parent {
    static {
        System.out.println("Child Static Block");
    }
}

public class Test {
    public static void main(String[] args) {
        Child c1 = new Child();
        Child c2 = new Child();
    }
}
```

Question:

- How many times will the static blocks be executed?
- Explain the order of execution.

Answer:

- **Output:**
- `Parent Static Block`
- `Child Static Block`
- **Explanation:**
    - Static blocks are executed **only once per class** during the **class loading** phase.
    - `Parent` is loaded first, followed by `Child`.
    - Creating `c1` and `c2` does not re-execute static blocks.

---

3. Abstract Class and Exception Propagation

```
abstract class Parent {
    abstract void calculate() throws Exception;
}

class Child extends Parent {
    void calculate() {
        System.out.println(10 / 0); // ArithmeticException
    }
}

public class Test {
    public static void main(String[] args) {
        Parent obj = new Child();
        try {
            obj.calculate();
        } catch (Exception e) {
            System.out.println("Caught Exception: " + e);
        }
    }
}
```

Question:

- Will this code compile? If not, why?
- If it compiles, what will be the output?

Answer:

- **Output:**
- `Exception in thread "main" java.lang.ArithmeticException: / by zero`
- **Explanation:**
    - The code compiles since `Child.calculate()` does not declare any checked exceptions.
    - However, it throws an **unchecked exception** (`ArithmeticException`) which is not caught.
    - The exception is propagated to the JVM, resulting in a runtime error.

---

4. Private Constructor with Factory Method

```
class Singleton {
    private static Singleton instance;
    private int value;

    private Singleton(int value) {
        this.value = value;
    }

    public static Singleton getInstance(int value) {
        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }

    public int getValue() {
        return value;
    }
}

public class Test {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance(10);
        Singleton s2 = Singleton.getInstance(20);
        System.out.println(s1.getValue() + " " + s2.getValue());
    }
}
```

Question:

- What will be the output?
- Explain why the second call does not change the value.

Answer:

- **Output:**
- `10 10`
- **Explanation:**
    - `Singleton` uses a **private constructor** and a **factory method** to ensure only one instance is created.

    - On the second call, `instance` is no longer `null`, so the existing object is returned with its initial value.

---

5. Nested Class and Shadowing

```
public class Outer {
    int x = 10;

    class Inner {
        int x = 20;
        void printValues(int x) {
            System.out.println("Local x: " + x);
            System.out.println("Inner x: " + this.x);
            System.out.println("Outer x: " + Outer.this.x);
        }
    }

    public static void main(String[] args) {
        Outer.Inner inner = new Outer().new Inner();
        inner.printValues(30);
    }
}
```

Question:

- What will be the output?
- Explain how variable shadowing works in this context.

Answer:

- **Output:**
- `Local x: 30`
- `Inner x: 20`
- `Outer x: 10`
- **Explanation:**
    - **Variable Shadowing** occurs when a variable in a local or inner scope hides a variable with the same name in an outer scope.
    - `this.x` refers to the inner class's variable, while `Outer.this.x` explicitly accesses the outer class variable.

---

6. Multiple Catch Blocks and Exception Hierarchy

```
public class Test {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[10] = 100;
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException Caught");
        } catch (Exception e) {
            System.out.println("General Exception Caught");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException Caught");
        }
    }
}
```

Question:

- Will this code compile? If not, why?
- If it compiles, what will be the output?

Answer:

- **Error:**
- `Unreachable catch block for ArrayIndexOutOfBoundsException.`
- **Explanation:**
    - Since `Exception` is a parent class of `ArrayIndexOutOfBoundsException`, placing it first makes all subsequent catch blocks **unreachable**.
    - To fix it, reorder the catch blocks from **specific to general**.

1. **List vs Set:**
   Explain a scenario where using a `Set` would be preferable over a `List`. Discuss the performance trade-offs when using `HashSet`, `LinkedHashSet`, and `TreeSet`.
2. **ConcurrentModificationException:**
   Consider the following code snippet:

```
List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C", "D"));
for (String s : list) {
    if (s.equals("B")) {
        list.remove(s);
    }
}
System.out.println(list);
```

   Explain what exception, if any, will occur and suggest ways to resolve it.

3. **Comparator vs Comparable:**
   Differentiate between `Comparator` and `Comparable` with appropriate examples. Why would you choose one over the other?
4. **HashMap Internal Working:**
   Explain how `HashMap` handles collisions internally. What happens when the number of entries exceeds the threshold of the map?
5. **Queue Implementation:**
   Describe a real-world scenario where a `PriorityQueue` is preferred over a `LinkedList` or `ArrayDeque`. Explain the underlying working of `PriorityQueue` in Java.
6. **Immutable Collections:**
   Explain the concept of immutable collections in Java using `Collections.unmodifiableList()` or `List.of()`. Provide a situation where using immutable collections would be beneficial.

---

1. **TreeMap and NavigableMap:**
   Implement a scenario using `TreeMap` where you need to efficiently fetch:
   - The greatest key less than a given key
   - The smallest key greater than a given key
     Explain how the `NavigableMap` interface supports such operations.
2. **Stream API and Parallel Processing:**
   Analyze the following code snippet:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
int sum = numbers.parallelStream()
                 .map(x -> x * 2)
                 .reduce(0, Integer::sum);
System.out.println(sum);
```

   - Explain how parallel streams function.
   - Would the result be consistent in all cases?
   - What are the potential drawbacks of using parallel streams?
3. **Custom Data Structure using LinkedList:**
   Design a custom data structure that uses a `LinkedList` internally to implement a

simplified version of a `Deque` (Double-Ended Queue). Provide the list of operations supported and their time complexity.

---

1. **Efficient Data Management:**
   You are designing a system to manage real-time log data from multiple servers. The logs arrive with a timestamp and are stored for quick access and analysis.
   - Which Java collection would you use for storing logs that need frequent insertion and deletion? Justify your choice.
   - Implement a mechanism to retrieve logs within a specified time range efficiently.
2. **Multi-level Sorting with Comparator:**
   You are tasked with sorting a list of `Employee` objects using the following criteria:
   - First by Department (Alphabetical)
   - Then by Age (Ascending)
   - Then by Name (Alphabetical)
     Write the required `Comparator` classes for implementing this multi-level sorting using Java Collections. Discuss the time complexity involved.

---

1. **List vs Set:**
   Answer:
   - Use a `Set` when you need to maintain unique elements.
   - `HashSet` is generally faster for inserts and lookups but is unordered.
   - `LinkedHashSet` maintains insertion order with slightly more memory overhead.
   - `TreeSet` maintains elements in a sorted order but has a higher time complexity (`O(log n)` for operations).

---

2. **ConcurrentModificationException:**
   Answer:
   - The code will throw a `ConcurrentModificationException` since you are modifying the list using `remove()` while iterating.
   - Solutions:
     - Use an `Iterator` and call `iterator.remove()` instead.
     - Use `List.removeIf()` for removal based on condition.
     - Use `CopyOnWriteArrayList` for concurrent modification.

---

3. **Comparator vs Comparable:**
   Answer:
   - `Comparable` is implemented by the class itself using the `compareTo()` method. It is used for natural ordering.
   - `Comparator` is a separate interface to define custom ordering. It allows sorting without modifying the class.

- Choose `Comparator` when you need multiple sorting criteria or if you cannot modify the source code.

---

4. **HashMap Internal Working:**
   Answer:
   - `HashMap` uses a combination of array and linked list (or red-black tree if buckets grow too large).
   - Collisions are handled using chaining or converting to a tree when the bucket size exceeds a threshold (typically 8).
   - When the size exceeds `LoadFactor * Capacity`, it resizes the array to maintain performance.

---

5. **Queue Implementation:**
   Answer:
   - Use a `PriorityQueue` in scenarios requiring ordered processing like job scheduling, Dijkstra's algorithm, or event management.
   - It uses a binary heap for efficient retrieval of the highest or lowest priority element in `O(log n)` time.

---

6. **Immutable Collections:**
   Answer:
   - Immutable collections ensure thread safety and protect data from accidental modification.
   - Example: Using `List.of()` or `Collections.unmodifiableList()` to return unmodifiable views.
   - Useful for maintaining application state, configurations, and constants.

---

1. **TreeMap and NavigableMap:**
   Answer:
   - `TreeMap` provides efficient lookup using methods like `lowerKey()`, `higherKey()`, `floorKey()`, and `ceilingKey()`.
   - It maintains keys in natural order using a Red-Black Tree.
   - Best suited for range-based queries and sorted data retrieval.

---

2. **Stream API and Parallel Processing:**
   Answer:
   - Parallel streams divide the task across multiple threads, leading to faster computation on large datasets.
   - However, for smaller datasets, thread management overhead may reduce performance.

   - Results are not guaranteed to be consistent in cases involving non-associative operations.

---

3. **Custom Data Structure using LinkedList:**
   Answer:
   - Implementing a custom `Deque` using a `LinkedList` would involve:
     - `addFirst()`, `addLast()` for adding elements.
     - `removeFirst()`, `removeLast()` for removals.
   - Time Complexity:
     - Insertion and Deletion: `O(1)`
     - Search: `O(n)`

---

1. **Efficient Data Management:**
   Answer:
   - Use a `TreeMap` or `PriorityQueue` for storing logs based on timestamps.
   - `TreeMap.subMap()` can retrieve logs within a specific time range in `O(log n)` time.
   - Maintain a secondary `HashMap` for storing logs by server ID.

---

2. **Multi-level Sorting with Comparator:**
   Answer:
   - Implement multiple `Comparator` classes using lambda functions or method references.
   - Chain them using `Comparator.thenComparing()` for multi-level sorting.
   - Time Complexity: `O(n log n)` using efficient sorting algorithms like TimSort.

---

1. **Why can't we create an object of an abstract class?**
   Answer:
   - Abstract classes are incomplete as they may contain abstract methods (methods without a body).
   - They are meant to be inherited and implemented by subclasses.
   - An abstract class serves as a blueprint for other classes.

---

2. **Explain the difference between fail-fast and fail-safe iterators. Provide examples.**
   Answer:
   - **Fail-Fast Iterators:** Detect structural modifications and throw `ConcurrentModificationException`. Examples: `ArrayList`, `HashMap`.
   - **Fail-Safe Iterators:** Do not throw exceptions, as they work on a copy of the collection. Examples: `CopyOnWriteArrayList`, `ConcurrentHashMap`.

---

3. **Why are wrapper classes immutable in Java?**
   Answer:
   - Wrapper classes (`Integer`, `Double`, etc.) are immutable to ensure security, thread safety, and caching optimization using the object pool.
   - This also allows them to be used as keys in collections like `HashMap` or `HashSet`.

---

4. **Explain how a private constructor is useful in design patterns.**
   Answer:
   - Private constructors restrict object creation.
   - Commonly used in **Singleton Design Pattern** to ensure only one instance exists.
   - Also used in **Factory Method Pattern** to control object creation using static methods.

---

5. **Compare Comparator and Comparable with real-world use cases.**
   Answer:
   - **Comparable:** Use it when the class has a natural sorting logic (e.g., `Student` by ID).
   - **Comparator:** Use it for multiple or dynamic sorting (e.g., sorting `Employee` by salary, name, or department using different Comparators).

---

6. **Can we override a private or static method in Java? Why or why not?**
   Answer:
   - Private methods cannot be overridden as they are not visible to subclasses.
   - Static methods belong to the class, not an instance. While they can be redefined in a subclass, it is method hiding, not overriding.

---

1. **Design an application using the Factory Design Pattern. Explain how it works.**
   Answer:
   - In a factory pattern, object creation logic is encapsulated in a factory class.
   - The client requests the object from the factory instead of using `new`.
   - Example: A vehicle factory that returns a `Car` or `Bike` object based on input.

---

2. **Explain how to prevent a class from being cloned.**
   Answer:
   - Implement the `Cloneable` interface and override the `clone()` method.
   - Throw `CloneNotSupportedException` in the `clone()` method.
   - Alternatively, declare the class `final` to prevent inheritance and cloning.

---

3. **What is the significance of the `volatile` keyword in Java?**
   Answer:
   - `volatile` ensures visibility of changes to variables across threads.
   - It prevents the compiler from optimizing variable access by storing it in a cache.
   - Useful in scenarios like implementing Singleton patterns with double-checked locking.

---

1. **Explain the Strategy Design Pattern with an example. How is it useful?**
   Answer:
   - Strategy Pattern defines a family of algorithms, encapsulates them, and makes them interchangeable.
   - Example: A payment gateway that supports different payment methods (`CreditCard`, `UPI`, `PayPal`).
   - The algorithm can be switched at runtime based on user selection.

---

2. **Describe the role of annotations in Java. How can custom annotations be used?**
   Answer:
   - Annotations provide metadata that can be used for code analysis, configuration, or processing at runtime.
   - Example: `@Override`, `@Deprecated`, and `@FunctionalInterface`.
   - Custom annotations can be created and processed using reflection via `AnnotationProcessor` for validations, logging, or frameworks like Spring.

---

1. **What is the difference between `Thread` class and `Runnable` interface? Which one is preferred and why?**
   Answer:
   - `Thread` class directly represents a thread of execution, while `Runnable` provides a task to be executed.
   - `Runnable` is preferred when the class needs to extend another class since Java supports single inheritance.

---

2. **Explain why the `start()` method is preferred over directly calling the `run()` method in Java threads.**
   Answer:
   - `start()` creates a new thread and calls the `run()` method asynchronously.
   - Calling `run()` directly executes it in the same thread, defeating the purpose of multithreading.

---

3. **What is a daemon thread? Provide a use case where daemon threads are useful.**
   Answer:
   - Daemon threads run in the background, providing services like garbage collection.
   - Example: A monitoring thread that logs system performance metrics.

---

4. **Explain the difference between `synchronized` block and `synchronized` method. Which one is more efficient?**
   Answer:
   - A `synchronized` block locks only a particular object, while a `synchronized` method locks the entire object or class.
   - `Synchronized` blocks are generally more efficient as they minimize the critical section.

---

5. **What are the differences between `wait()`, `notify()`, and `notifyAll()` in Java?**
   Answer:
   - `wait()` releases the lock and puts the thread in a waiting state.
   - `notify()` wakes up a single waiting thread.
   - `notifyAll()` wakes up all waiting threads on the same object.

---

6. **Can a thread be restarted once it has completed its execution? Justify your answer.**
   Answer:
   - No, a thread cannot be restarted once it is terminated (`Thread.State.TERMINATED`).
   - A new instance of the thread must be created instead.

---

1. **Describe the concept of thread starvation and how it can occur in a Java application.**
   Answer:
   - Thread starvation occurs when low-priority threads are unable to gain CPU time due to high-priority threads monopolizing resources.
   - It can be mitigated using techniques like fair scheduling or adjusting priorities.

---

2. **Explain how the `ReentrantLock` is different from the `synchronized` keyword.**
   Answer:
   - `ReentrantLock` provides more advanced locking mechanisms like fairness policies, timed locks, and interruptible locks.
   - It is preferred when additional flexibility in thread control is needed.

---

3. **What is a thread pool and why is it beneficial?**
   Answer:
   - A thread pool manages a fixed number of reusable threads to execute tasks efficiently.
   - It reduces overhead caused by frequent thread creation and destruction, improving performance.

---

1. **Design a scenario where two threads need to print even and odd numbers alternatively using `wait()` and `notify()`. Explain your solution.**
   Answer:
   - Create two threads: one for even numbers and one for odd numbers.
   - Use a shared object and synchronized block.
   - Each thread calls `wait()` when the number to print doesn't match its assigned task.
   - Use `notify()` to wake up the waiting thread when it's the correct turn.

---

2. **Explain the concept of deadlock with an example. Suggest strategies to avoid it.**
   Answer:
   - Deadlock occurs when two or more threads are blocked forever, each waiting for a resource held by another thread.
   - Example: Thread A holds Resource 1 and waits for Resource 2, while Thread B holds Resource 2 and waits for Resource 1.
   - Prevention strategies:
     - Acquire locks in a consistent order.
     - Use timeouts with `tryLock()`.
     - Avoid nested locks when possible.

Question 1: Deadlock Scenario

What will happen when the following code is executed? Identify the issue and suggest a fix.

```
class Resource {
    void methodA() {
        synchronized (this) {
            System.out.println("Method A");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
        }
    }
    void methodB() {
        synchronized (this) {
            System.out.println("Method B");
        }
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        Resource resource = new Resource();

        Thread t1 = new Thread(() -> resource.methodA());
        Thread t2 = new Thread(() -> resource.methodB());

        t1.start();
        t2.start();
    }
}
```

Problem:

- The code might cause a deadlock if additional locks are introduced within `methodA` or `methodB` in a complex scenario.
- Single lock on `this` prevents true concurrency.

Solution:

- Introduce fine-grained locking using separate objects.
- Example:

```
private final Object lockA = new Object();
private final Object lockB = new Object();
```

- Use these for synchronized blocks instead of locking on `this`.

Question 2: Thread Starvation

What's wrong with this code, and how will it behave? How can you resolve the issue?

```
public class StarvationExample {
    public static void main(String[] args) {
        Runnable task = () -> {
            synchronized (StarvationExample.class) {
                System.out.println(Thread.currentThread().getName() + "
acquired lock.");
                try { Thread.sleep(5000); } catch (InterruptedException e)
{}
            }
        };

        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(task, "Thread-" + i);
            if (i == 4) t.setPriority(Thread.MAX_PRIORITY);
            t.start();
        }
    }
}
```

Problem:

- `Thread-4` has maximum priority and can prevent low-priority threads from executing.
- Results in **Thread Starvation**.

Solution:

- Avoid using thread priority for critical code.
- Use proper synchronization and ensure fairness with `ReentrantLock` using `new ReentrantLock(true)`.

Question 3: Incorrect Use of `wait()` and `notify()`

Identify the logical flaw in the following code:

```
class SharedResource {
    synchronized void printMessage() throws InterruptedException {
        System.out.println("Waiting for signal...");
        wait();
        System.out.println("Signal received. Resuming execution.");
    }

    synchronized void sendSignal() {
        System.out.println("Sending signal...");
        notify();
    }
}

public class NotifyExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        new Thread(() -> {
            try {
                resource.printMessage();
            } catch (InterruptedException e) {}
        }).start();

        new Thread(() -> resource.sendSignal()).start();
    }
}
```

Problem:

- The second thread may call `notify()` before the first thread calls `wait()`, causing a missed signal.
- **Race Condition** occurs.

Solution:

- Use a flag with a `while` loop to ensure `wait()` works correctly:

```
boolean signalSent = false;
synchronized void printMessage() throws InterruptedException {
    while (!signalSent) {
        wait();
    }
    System.out.println("Signal received. Resuming execution.");
}
```

- After sending the signal, set `signalSent = true;`.

Question 4: Improper Thread Interruption

What issue will arise from this code? How can it be fixed?

```
public class InterruptionExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            while (true) {
                System.out.println("Thread running...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("Interrupted!");
                }
            }
        });

        thread.start();
        try { Thread.sleep(3000); } catch (InterruptedException e) {}
        thread.interrupt();
    }
}
```

Problem:

- The `catch` block handles the exception, but the thread resumes its infinite loop without stopping.
- The thread will not terminate.

Solution:

- Check the interrupt status using `Thread.currentThread().isInterrupted()` in the loop:

```
while (!Thread.currentThread().isInterrupted()) {
    System.out.println("Thread running...");
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        System.out.println("Interrupted!");
        break;
    }
}
```

Question 1: Misuse of HashMap with Mutable Keys

What issue will this code face, and how can it be fixed?

```
import java.util.HashMap;
import java.util.Map;

class Employee {
    String name;
    int id;

    Employee(String name, int id) {
        this.name = name;
        this.id = id;
    }

    @Override
    public String toString() {
        return name + " (ID: " + id + ")";
    }
}

public class HashMapIssue {
    public static void main(String[] args) {
        Map<Employee, String> map = new HashMap<>();
        Employee e1 = new Employee("Alice", 101);
        Employee e2 = new Employee("Bob", 102);

        map.put(e1, "Developer");
        map.put(e2, "Manager");

        System.out.println("Before modification: " + map.get(e1));

        e1.id = 201; // Key is modified

        System.out.println("After modification: " + map.get(e1));
    }
}
```

Problem:

- The `HashMap` relies on `hashCode()` and `equals()` for key lookups.
- Modifying key fields like `e1.id` results in unpredictable behavior and key loss.

Solution:

- Make the fields immutable using `final`.
- Implement `hashCode()` and `equals()` correctly.
- Example:

```
@Override
public int hashCode() {
    return Integer.hashCode(id);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    Employee other = (Employee) obj;
    return id == other.id && name.equals(other.name);
}
```

Question 2: ConcurrentModificationException

Identify the issue and suggest a fix.

```
import java.util.ArrayList;
import java.util.List;

public class ConcurrentModificationExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        for (String fruit : list) {
            if (fruit.equals("Banana")) {
                list.remove(fruit);
            }
        }

        System.out.println("List after removal: " + list);
    }
}
```

Problem:

- This code will throw a `ConcurrentModificationException` because you are modifying the list while iterating using an enhanced for-loop.

Solution:

- Use an `Iterator` instead:

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    if (iterator.next().equals("Banana")) {
        iterator.remove();
    }
}
```

- Alternatively, use `removeIf()` from Java 8:

```
list.removeIf(fruit -> fruit.equals("Banana"));
```

Question 3: PriorityQueue Misuse

What's wrong with the output of this code?

```
import java.util.PriorityQueue;
```

```
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.add(10);
        pq.add(50);
        pq.add(30);
        pq.add(5);

        System.out.println("Peek: " + pq.peek());
        System.out.println("Poll: " + pq.poll());
        System.out.println("Remaining Queue: " + pq);
    }
}
```

Problem:

- `PriorityQueue` does not maintain the insertion order.
- It orders the elements based on **natural ordering** or a **custom comparator**.

Solution:

- If a specific order is needed, use a `Comparator` or switch to a `LinkedList` or `ArrayList` instead.
- Example using a custom comparator:

```
PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> b - a);
```

This makes it a **Max-Heap**.

Question 4: LinkedList Misinterpretation

What will be the output of the following code? Explain the behavior.

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        list.add(10);
        list.add(20);
        list.addFirst(5);
        list.addLast(30);
        list.removeFirst();
        list.removeLast();

        System.out.println(list);
    }
}
```

Answer:

- Output: `[10, 20]`
- Explanation:

    - `addFirst(5)` inserts at the beginning.
    - `addLast(30)` inserts at the end.
    - `removeFirst()` removes `5`.
    - `removeLast()` removes `30`.

Question 5: Misusing Comparable and Comparator

What's wrong with this code? Why doesn't it sort as expected?

```
import java.util.*;

class Student implements Comparable<Student> {
    String name;
    int marks;

    Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    @Override
    public int compareTo(Student other) {
        return marks - other.marks;
    }

    @Override
    public String toString() {
        return name + " (" + marks + ")";
    }
}

public class SortingIssue {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 85));
        students.add(new Student("Bob", 92));
        students.add(new Student("Charlie", 85));
        students.add(new Student("David", 80));

        Collections.sort(students);
        System.out.println(students);
    }
}
```

Problem:

- The `compareTo()` method only compares based on marks.
- If two students have the same marks, they will not be sorted by name, violating consistent sorting behavior.

Solution:

- Update the `compareTo()` method to include a secondary sorting criterion using `name`:

```
@Override
```

```
public int compareTo(Student other) {
    if (marks == other.marks) {
        return name.compareTo(other.name);
    }
    return marks - other.marks;
}
```

- Alternatively, use a custom `Comparator` for sorting.

Question 6: Incorrect TreeSet Behavior

Why does this code not print all the elements?

```
import java.util.*;

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class TreeSetIssue {
    public static void main(String[] args) {
        Set<Person> set = new TreeSet<>();
        set.add(new Person("Alice", 30));
        set.add(new Person("Bob", 25));
        set.add(new Person("Charlie", 30));
        System.out.println(set);
    }
}
```

Problem:

- `TreeSet` requires elements to be comparable using `Comparable` or `Comparator`.
- Since the `Person` class does not implement `Comparable`, this will throw a `ClassCastException`.

Solution:

- Implement `Comparable` or use a `Comparator`:

```
Set<Person> set = new TreeSet<>((p1, p2) -> Integer.compare(p1.age,
p2.age));
```

- Now it will work without exceptions.

Question 7: Unexpected Behavior with HashSet

Why does the following code produce unexpected results?

```
import java.util.*;

public class HashSetIssue {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("apple");
        set.add("banana");
        set.add("apple");
        set.add("cherry");

        System.out.println(set);
    }
}
```

Problem:

- `HashSet` does not allow duplicates, but the above code may sometimes fail to detect duplicates if the `hashCode()` method behaves incorrectly.
- However, in this case, since `String` class overrides `hashCode()` and `equals()`, the code works as expected.

Solution:

- No solution is needed for `String`.
- For custom objects, always ensure proper implementation of `equals()` and `hashCode()`.

Question 8: Incorrect Usage of ListIterator

Identify the issue in this code.

```
import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C",
"D"));

        ListIterator<String> iterator = list.listIterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
            if (iterator.nextIndex() == 2) {
                iterator.add("X");
            }
        }
        System.out.println("\nUpdated List: " + list);
    }
}
```

Problem:

- The call `iterator.add("X")` will insert `"X"` at index 2. However, using `add()` without considering the index can cause index confusion and inconsistent results.

Solution:

- Ensure the `add()` call is carefully placed and the `next()` is used correctly.
- Example Fix:

```
if (iterator.previousIndex() == 1) {
    iterator.add("X");
}
```

Question 9: Using Queue Incorrectly

What will be the output of the following code?

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(10);
        queue.add(20);
        queue.add(30);

        System.out.println("Head of queue: " + queue.peek());
        System.out.println("Removing: " + queue.poll());
        System.out.println("Head after poll: " + queue.peek());
        queue.remove();
        System.out.println("Final Queue: " + queue);
    }
}
```

Answer:

- Output:

```
Head of queue: 10
Removing: 10
Head after poll: 20
Final Queue: [30]
```

- Explanation:
  - `peek()` returns the head of the queue without removing it.
  - `poll()` removes and returns the head.
  - `remove()` removes the next head.

Question 1: File Not Found Exception

What is wrong with this code?

```
import java.io.*;

public class FileExample {
    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader("nonexistentfile.txt");
            BufferedReader br = new BufferedReader(reader);
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Problem:

- The file `"nonexistentfile.txt"` doesn't exist, leading to a `FileNotFoundException`.

Solution:

- Ensure the file exists or use appropriate error handling to inform the user.
- You can check for file existence before reading using `file.exists()`.

Question 2: Resource Leak Issue

What is the issue in the following code?

```
import java.io.*;

public class ResourceLeakExample {
    public static void main(String[] args) throws IOException {
        FileWriter writer = new FileWriter("output.txt");
        writer.write("Hello, World!");
        System.out.println("Message Written!");
    }
}
```

Problem:

- The `FileWriter` is not closed, leading to a resource leak.

Solution:

- Use `try-with-resources` to ensure resources are closed automatically.

```
try (FileWriter writer = new FileWriter("output.txt")) {
    writer.write("Hello, World!");
}
```

Question 3: File Reading Issue

What is the output of this code if the file contains multiple lines?

```
import java.io.*;

public class FileReadExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("data.txt"))) {
            System.out.println(br.readLine());
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Problem:

- The code reads only the first two lines using `readLine()`.

Solution:

- To read the entire file, use a loop.

```
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```

Question 4: File Overwriting vs. Appending

What happens if this code is executed multiple times?

```
import java.io.*;

public class FileOverwriteExample {
    public static void main(String[] args) throws IOException {
        FileWriter writer = new FileWriter("log.txt");
        writer.write("Session Started\n");
        writer.close();
    }
}
```

Problem:

- Every time the program runs, it overwrites the contents of `log.txt`.

Solution:

- Use the `FileWriter` constructor with `true` to append data instead.

```
FileWriter writer = new FileWriter("log.txt", true);
```

```
writer.write("Session Started\n");
```

Question 5: Incorrect Path Issue

What is the issue with the following code?

```
import java.io.*;

public class PathIssueExample {
    public static void main(String[] args) {
        File file = new File("documents/output.txt");
        try (FileWriter writer = new FileWriter(file)) {
            writer.write("Hello, File!");
        } catch (IOException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Problem:

- The directory `documents` may not exist, causing a `FileNotFoundException`.

Solution:

- Ensure the directories exist using `mkdirs()`.

```
file.getParentFile().mkdirs();
```

This will create missing parent directories before writing to the file.

Question 6: File Deletion Issue

What is the issue with the following code?

```
import java.io.*;

public class DeleteFileExample {
    public static void main(String[] args) {
        File file = new File("sample.txt");
        if (file.delete()) {
            System.out.println("File deleted successfully.");
        } else {
            System.out.println("File deletion failed.");
        }
    }
}
```

Problem:

- The file might not exist or could be locked by another process.
- Lack of permissions could also cause failure.

Solution:

- Check if the file exists before attempting deletion using `file.exists()`.
- Ensure the file is not being used elsewhere.

```
if (file.exists()) {
    if (file.delete()) {
        System.out.println("File deleted successfully.");
    } else {
        System.out.println("File deletion failed.");
    }
} else {
    System.out.println("File not found.");
}
```

Question 7: File Copy Issue

What is wrong with the following file copy logic?

```
import java.io.*;

public class FileCopyExample {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("source.txt");
        FileOutputStream fos = new FileOutputStream("destination.txt");
        int data;
        while ((data = fis.read()) != -1) {
            fos.write(data);
        }
        System.out.println("File copied successfully.");
    }
}
```

Problem:

- The resources are not closed, which may lead to data loss or memory leaks.

Solution:

- Use `try-with-resources` for proper resource management.

```
try (FileInputStream fis = new FileInputStream("source.txt");
     FileOutputStream fos = new FileOutputStream("destination.txt")) {
    int data;
    while ((data = fis.read()) != -1) {
        fos.write(data);
    }
    System.out.println("File copied successfully.");
}
```

Question 8: Buffered Reader vs File Reader

Why would the following code be inefficient for large files?

```
import java.io.*;

public class InefficientReadExample {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("largefile.txt");
        int data;
        while ((data = fr.read()) != -1) {
            System.out.print((char) data);
        }
    }
}
```

Problem:

- `FileReader.read()` reads one character at a time, which is slow for large files.

Solution:

- Use `BufferedReader` for efficient reading.

```
BufferedReader br = new BufferedReader(new FileReader("largefile.txt"));
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```

Question 9: File Append Issue

What will be the output of the following code if `"data.txt"` contains `Hello`?

```
import java.io.*;

public class AppendExample {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("data.txt", true);
        fw.write(" World");
        fw.close();
    }
}
```

Answer:

- The code will append `" World"` to the existing content, resulting in:

```
Hello World
```

Question 10: File Handling with Exception Handling

What is the problem with the exception handling in this code?

```
import java.io.*;

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        File file = new File("test.txt");
        try {
            FileReader reader = new FileReader(file);
            System.out.println("File opened successfully.");
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
        } catch (IOException e) {
            System.out.println("An IO error occurred.");
        }
    }
}
```

Problem:

- `FileReader` only throws `FileNotFoundException`, and the `IOException` block is unnecessary.

Solution:

- Remove the `IOException` catch block.

```
catch (FileNotFoundException e) {
    System.out.println("File not found.");
}
```

Alternatively, you can keep the `IOException` block if you handle reading operations later.

Question 1: Misuse of Generics

What is the problem with the following code?

```
import java.util.*;

public class GenericsExample {
    public static void main(String[] args) {
        List list = new ArrayList();
        list.add("Hello");
        list.add(123);
        System.out.println(list);
    }
}
```

Problem:

- No generics are used, which means the list accepts any object.
- Type safety is compromised, and runtime exceptions may occur during operations.

Solution:

- Use proper generics to ensure type safety.

```
List<String> list = new ArrayList<>();
list.add("Hello");
// list.add(123); // This will give a compile-time error
```

Question 2: Incorrect Sorting

Identify the issue in the sorting logic of this code:

```
import java.util.*;

public class SortExample {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(4, 2, 8, 5);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Problem:

- `Arrays.asList()` returns a fixed-size list that cannot be resized or modified.
- Sorting it directly can cause an `UnsupportedOperationException` depending on further operations.

Solution:

- Convert to a mutable list before sorting.

```
List<Integer> list = new ArrayList<>(Arrays.asList(4, 2, 8, 5));
```

```
Collections.sort(list);
System.out.println(list); // Output: [2, 4, 5, 8]
```

Question 3: Incorrect Use of HashMap

What's wrong with this code using `HashMap`?

```
import java.util.*;

public class HashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("A", 1);
        map.put("B", 2);
        map.put("A", 3);
        System.out.println(map);
    }
}
```

Answer:

- `HashMap` allows only unique keys.
- The second `"A"` overwrites the first value, resulting in `{A=3, B=2}`.

Question 4: NullPointerException in TreeSet

Why does this code throw a `NullPointerException`?

```
import java.util.*;

public class TreeSetExample {
    public static void main(String[] args) {
        Set<String> set = new TreeSet<>();
        set.add(null);
    }
}
```

Problem:

- `TreeSet` uses a comparator for sorting elements.
- Null values are not allowed as it cannot compare null with other elements.

Solution:

- Use a `HashSet` if you need to store `null` values.

```
Set<String> set = new HashSet<>();
set.add(null);
```

Question 5: Generics with Wildcards

What will be the output of this code?

```
import java.util.*;

public class WildcardExample {
    public static void printList(List<?> list) {
        System.out.println(list);
    }

    public static void main(String[] args) {
        List<String> strList = Arrays.asList("A", "B", "C");
        List<Integer> intList = Arrays.asList(1, 2, 3);

        printList(strList);
        printList(intList);
    }
}
```

Answer:

- The code will print both lists successfully:

```
[A, B, C]
[1, 2, 3]
```

- `List<?>` uses an unbounded wildcard to accept any type of list.

Question 6: ConcurrentModificationException

What is wrong with this code?

```
import java.util.*;

public class ConcurrentExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
        for (String s : list) {
            if (s.equals("B")) {
                list.remove(s);
            }
        }
        System.out.println(list);
    }
}
```

Problem:

- Removing elements directly from the list during iteration causes a `ConcurrentModificationException`.

Solution:

- Use an `Iterator` for safe removal.

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    if (iterator.next().equals("B")) {
        iterator.remove();
    }
}
```

Question 7: PriorityQueue with Custom Comparator

Why does this code produce unexpected output?

```
import java.util.*;

public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<String> pq = new PriorityQueue<>((a, b) ->
b.compareTo(a));
        pq.add("Apple");
        pq.add("Banana");
        pq.add("Cherry");

        while (!pq.isEmpty()) {
            System.out.println(pq.poll());
        }
    }
}
```

Answer:

- The priority queue uses a comparator for reverse sorting.
- The output will be:

```
Cherry
Banana
Apple
```

- `b.compareTo(a)` causes the queue to behave as a max-heap.