

Java singleton and immutable classes

java programming questions and answers

give complete all types to create singleton and how to use

how many ways we can create immutable class explain with code

and give more promising questions and answers

Alright, let's make this a **Java interview booster pack** for you.

I'll cover:

1. All types of Singleton implementations + usage examples.
2. Ways to create Immutable classes with examples.
3. More important Java programming questions with answers.

1. All Ways to Create a Singleton in Java

A **Singleton** ensures that **only one instance** of a class exists and provides a global point of access to it.

A. Eager Initialization

```
java

public class EagerSingleton {
    private static final EagerSingleton instance = new
    EagerSingleton();

    private EagerSingleton() {}

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

✅ **Pros:** Simple, thread-safe without synchronization.

❌ **Cons:** Instance created even if never used.

B. Lazy Initialization (Not Thread-Safe)

```
java

public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {}

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

✗ Not safe in multi-threaded environments.

C. Thread-Safe Singleton (Synchronized Method)

```
java

public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance;

    private ThreadSafeSingleton() {}

    public static synchronized ThreadSafeSingleton
    getInstance() {
        if (instance == null) {
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }
}
```

✓ Safe, but synchronization has performance cost.

D. Double-Checked Locking (Best for Performance)

```
java
```

```
public class DoubleCheckedSingleton {
    private static volatile DoubleCheckedSingleton instance;

    private DoubleCheckedSingleton() {}

    public static DoubleCheckedSingleton getInstance() {
        if (instance == null) {
            synchronized (DoubleCheckedSingleton.class) {
                if (instance == null) {
                    instance = new DoubleCheckedSingleton();
                }
            }
        }
        return instance;
    }
}
```

✅ Thread-safe and efficient.

E. Bill Pugh Singleton (Inner Static Helper Class)

```
java

public class BillPughSingleton {
    private BillPughSingleton() {}

    private static class Helper {
        private static final BillPughSingleton INSTANCE = new
BillPughSingleton();
    }

    public static BillPughSingleton getInstance() {
        return Helper.INSTANCE;
    }
}
```

✅ Thread-safe, lazy-loaded, and no synchronization overhead.

F. Enum Singleton (Best & Serialization Safe)

```
java
```

```
public enum EnumSingleton {  
    INSTANCE;  
  
    public void showMessage() {  
        System.out.println("Hello from Enum Singleton!");  
    }  
}
```

- ✅ Serialization-safe, thread-safe, prevents reflection attacks.
- ❌ Not flexible if you want to lazily load with parameters.

Usage:

```
java  
  
public class Main {  
    public static void main(String[] args) {  
        EnumSingleton.INSTANCE.showMessage();  
    }  
}
```

2. How to Create an Immutable Class in Java

Immutable Class Rules:

1. Mark class as `final` (optional, but prevents subclassing).
2. Make all fields `private final`.
3. No setters.
4. Initialize fields only via constructor.
5. For mutable fields (like `Date`, collections), use deep copy in getters and constructors.

A. Simple Immutable Class

```
java  
  
public final class ImmutablePerson {  
    private final String name;  
    private final int age;
```

```
public ImmutablePerson(String name, int age) {  
    this.name = name;  
    this.age = age;  
}  
  
public String getName() { return name; }  
public int getAge() { return age; }  
}
```

B. Immutable Class with Mutable Field

```
java  
  
import java.util.Date;  
  
public final class ImmutableEvent {  
    private final String title;  
    private final Date date;  
  
    public ImmutableEvent(String title, Date date) {  
        this.title = title;  
        this.date = new Date(date.getTime()); // Deep copy  
    }  
  
    public String getTitle() { return title; }  
  
    public Date getDate() {  
        return new Date(date.getTime()); // Deep copy  
    }  
}
```

Ways to Create Immutable Class

1. Simple final fields (e.g., `String`, `Integer`).
 2. Defensive copies for mutable objects.
 3. Use unmodifiable collections from `Collections.unmodifiableList()`.
 4. Use records (Java 14+).
 5. Builder pattern returning immutable objects.
-

3. More Java Programming Interview Q&A

Q1: Difference between `==` and `.equals()`

- `==` → Compares references.
- `.equals()` → Compares values (can be overridden).

```
java
```

```
String s1 = new String("hello");
String s2 = new String("hello");
System.out.println(s1 == s2);      // false
System.out.println(s1.equals(s2)); // true
```

Q2: Explain `final`, `finally`, and `finalize()`

- `final` → Constant variables, prevent method overriding, prevent inheritance.
- `finally` → Block that always executes after try/catch.
- `finalize()` → Called by GC before object is destroyed (deprecated).

Q3: Why String is Immutable in Java?

- Security (used in class loaders, network connections).
- Thread-safety.
- Caching & performance.

Q4: Difference between `HashMap` and `ConcurrentHashMap`

- `HashMap` is not thread-safe.
- `ConcurrentHashMap` allows concurrent reads and thread-safe writes.

Q5: What is Java Memory Model (JMM)?

Defines how threads interact through memory, guaranteeing visibility of changes and ordering of operations.

Q6: Explain `volatile` keyword

Ensures visibility of changes across threads and prevents instruction reordering.

Q7: Difference between `sleep()` and `wait()`

- `sleep()` → Static method in `Thread`, does not release lock.
 - `wait()` → Instance method in `Object`, releases lock and waits for notification.
-

Q8: Functional Interface & Example

An interface with **only one abstract method**.

```
java

@FunctionalInterface
interface MyFunction {
    void execute();
}
```

If you want, I can **extend this with 50+ more advanced Java interview questions** focusing on concurrency, JVM internals, collections, design patterns, and real-world coding examples.

That way you'll have a **complete preparation guide**.

Do you want me to prepare that full list?

i need java programs with code explain

Got it — you don't just want definitions and theory, you want **Java programs with code + explanations** so you can see them in action.

I'll start by giving you a **collection of core Java programming examples** (with explanations), including:

1. **Singleton pattern programs** (all types, runnable).
 2. **Immutable class programs** (runnable).
 3. **Additional common Java programs** asked in interviews.
-

1. Singleton Pattern – All Implementations with Usage

A. Eager Initialization

```
java

// EagerSingleton.java
public class EagerSingleton {
    private static final EagerSingleton instance = new
    EagerSingleton();

    // private constructor prevents instantiation
    private EagerSingleton() {}

    public static EagerSingleton getInstance() {
        return instance;
    }

    public void showMessage() {
        System.out.println("Eager Singleton Instance");
    }

    public static void main(String[] args) {
        EagerSingleton obj1 = EagerSingleton.getInstance();
        EagerSingleton obj2 = EagerSingleton.getInstance();

        obj1.showMessage();
        System.out.println(obj1 == obj2); // true
    }
}
```

Explanation:

- Instance created at class loading time.
- Thread-safe without synchronization.
- Wastes memory if never used.

B. Lazy Initialization (Thread-Safe)

```
java

public class LazySingleton {
    private static LazySingleton instance;
```



```

private LazySingleton() {}

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

public static void main(String[] args) {
    LazySingleton obj1 = LazySingleton.getInstance();
    LazySingleton obj2 = LazySingleton.getInstance();
    System.out.println(obj1 == obj2); // true
}
}

```

Explanation:

- Instance created only when first needed.
- Synchronization ensures thread safety, but can be slower.

C. Double-Checked Locking

```

java

public class DoubleCheckedSingleton {
    private static volatile DoubleCheckedSingleton instance;

    private DoubleCheckedSingleton() {}

    public static DoubleCheckedSingleton getInstance() {
        if (instance == null) {
            synchronized (DoubleCheckedSingleton.class) {
                if (instance == null) {
                    instance = new DoubleCheckedSingleton();
                }
            }
        }
        return instance;
    }

    public static void main(String[] args) {
        DoubleCheckedSingleton obj1 =
        DoubleCheckedSingleton.getInstance();
        DoubleCheckedSingleton obj2 =

```

```

DoubleCheckedSingleton.getInstance();
    System.out.println(obj1 == obj2); // true
}
}

```

Explanation:

- Uses `volatile` to prevent instruction reordering.
- Synchronization only happens once during initialization.

D. Bill Pugh Singleton

```

java

public class BillPughSingleton {
    private BillPughSingleton() {}

    private static class Helper {
        private static final BillPughSingleton INSTANCE = new
BillPughSingleton();
    }

    public static BillPughSingleton getInstance() {
        return Helper.INSTANCE;
    }

    public static void main(String[] args) {
        BillPughSingleton obj1 =
BillPughSingleton.getInstance();
        BillPughSingleton obj2 =
BillPughSingleton.getInstance();
        System.out.println(obj1 == obj2); // true
    }
}

```

Explanation:

- Inner static helper class creates instance only when needed.
- No synchronization overhead.

E. Enum Singleton

```

java

```

```
public enum EnumSingleton {  
    INSTANCE;  
  
    public void show() {  
        System.out.println("Enum Singleton Instance");  
    }  
  
    public static void main(String[] args) {  
        EnumSingleton obj1 = EnumSingleton.INSTANCE;  
        EnumSingleton obj2 = EnumSingleton.INSTANCE;  
        obj1.show();  
        System.out.println(obj1 == obj2); // true  
    }  
}
```

Explanation:

- Serialization-safe and reflection-proof.

2. Immutable Class Examples

A. Simple Immutable Class

```
java  
  
public final class ImmutablePerson {  
    private final String name;  
    private final int age;  
  
    public ImmutablePerson(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
    public int getAge() { return age; }  
  
    public static void main(String[] args) {  
        ImmutablePerson p = new ImmutablePerson("Alice", 30);  
        System.out.println(p.getName() + " - " + p.getAge());  
    }  
}
```

Explanation:

- Class is `final`.
 - All fields are `private final`.
 - No setters.
-

B. Immutable Class with Mutable Field

```
java

import java.util.Date;

public final class ImmutableEvent {
    private final String title;
    private final Date date;

    public ImmutableEvent(String title, Date date) {
        this.title = title;
        this.date = new Date(date.getTime()); // deep copy
    }

    public String getTitle() { return title; }

    public Date getDate() {
        return new Date(date.getTime()); // deep copy
    }

    public static void main(String[] args) {
        Date d = new Date();
        ImmutableEvent e = new ImmutableEvent("Meeting", d);

        System.out.println("Original: " + e.getDate());
        d.setTime(0); // trying to modify original date
        System.out.println("After change: " + e.getDate());
    }
}
```

Explanation:

- Defensively copies mutable objects in constructor & getters.
-

3. More Common Java Programs for Interviews

A. Reverse a String

```
java

public class ReverseString {
    public static void main(String[] args) {
        String str = "Java";
        String reversed = new
StringBuilder(str).reverse().toString();
        System.out.println(reversed);
    }
}
```

B. Check Palindrome

```
java

public class PalindromeCheck {
    public static void main(String[] args) {
        String str = "madam";
        String rev = new
StringBuilder(str).reverse().toString();
        System.out.println(str.equals(rev) ? "Palindrome" :
"Not Palindrome");
    }
}
```

C. Factorial Using Recursion

```
java

public class Factorial {
    public static int fact(int n) {
        if (n <= 1) return 1;
        return n * fact(n - 1);
    }

    public static void main(String[] args) {
        System.out.println(fact(5)); // 120
    }
}
```

D. Fibonacci Series

```
java

public class Fibonacci {
    public static void main(String[] args) {
        int n = 10, first = 0, second = 1;
        System.out.print(first + " " + second);
        for (int i = 2; i < n; i++) {
            int next = first + second;
            System.out.print(" " + next);
            first = second;
            second = next;
        }
    }
}
```

If you want, I can prepare a **full Java program bank** with **50+ runnable codes** (patterns, collections, multithreading, file handling, OOP, etc.) — so you can **practice and run them directly** in your IDE without searching online.

Do you want me to prepare that next? That would be like your **Java coding master kit** for interviews.

Java coding master kit for interviews.

Alright — let's build your **Java Coding Master Kit for Interviews** 🚀.

This will be **practical, runnable, and complete with explanations** so you can copy-paste into your IDE and start practicing immediately.

I'll break it into **10 major categories** so you cover **core + advanced** topics:

Java Coding Master Kit

(50+ Programs with Explanations)

Category 1 – Basics & Strings

1. Reverse a String

```
java

public class ReverseString {
    public static void main(String[] args) {
        String str = "Java";
        String reversed = new
StringBuilder(str).reverse().toString();
        System.out.println(reversed);
    }
}
```

2. Check Palindrome

```
java

public class PalindromeCheck {
    public static void main(String[] args) {
        String str = "madam";
        String rev = new
StringBuilder(str).reverse().toString();
        System.out.println(str.equals(rev) ? "Palindrome" :
"Not Palindrome");
    }
}
```

3. Count Vowels & Consonants

```
java

public class VowelConsonantCount {
    public static void main(String[] args) {
        String str = "Java Programming".toLowerCase();
        int vowels = 0, consonants = 0;

        for (char c : str.toCharArray()) {
            if (c >= 'a' && c <= 'z') {
                if ("aeiou".indexOf(c) != -1)
```

```

        vowels++;
    }
    else
        consonants++;
    }
}
System.out.println("Vowels: " + vowels + ",
Consonants: " + consonants);
}
}

```

4. Anagram Check

```

java

import java.util.Arrays;

public class AnagramCheck {
    public static void main(String[] args) {
        String s1 = "listen", s2 = "silent";
        char[] a = s1.toCharArray();
        char[] b = s2.toCharArray();
        Arrays.sort(a);
        Arrays.sort(b);
        System.out.println(Arrays.equals(a, b) ? "Anagram" :
"Not Anagram");
    }
}

```

5. Remove Duplicates from String

```

java

public class RemoveDuplicates {
    public static void main(String[] args) {
        String str = "programming";
        StringBuilder sb = new StringBuilder();
        for (char c : str.toCharArray()) {
            if (sb.indexOf(String.valueOf(c)) == -1)
                sb.append(c);
        }
        System.out.println(sb.toString());
    }
}

```



```
}  
}
```

Category 2 – Numbers

6. Factorial (Recursion)

java

```
public class Factorial {  
    public static int fact(int n) {  
        if (n <= 1) return 1;  
        return n * fact(n - 1);  
    }  
    public static void main(String[] args) {  
        System.out.println(fact(5));  
    }  
}
```

7. Fibonacci Series

java

```
public class Fibonacci {  
    public static void main(String[] args) {  
        int n = 10, first = 0, second = 1;  
        System.out.print(first + " " + second);  
        for (int i = 2; i < n; i++) {  
            int next = first + second;  
            System.out.print(" " + next);  
            first = second;  
            second = next;  
        }  
    }  
}
```

8. Prime Number Check

java

```
public class PrimeCheck {  
    public static void main(String[] args) {  
        int num = 29;  
        boolean prime = num > 1;  
        for (int i = 2; i <= Math.sqrt(num); i++) {  
            if (num % i == 0) {  
                prime = false;  
                break;  
            }  
        }  
        System.out.println(prime ? "Prime" : "Not Prime");  
    }  
}
```

9. Armstrong Number

java

```
public class Armstrong {  
    public static void main(String[] args) {  
        int num = 153, sum = 0, temp = num;  
        while (temp > 0) {  
            int digit = temp % 10;  
            sum += Math.pow(digit, 3);  
            temp /= 10;  
        }  
        System.out.println(sum == num ? "Armstrong" : "Not  
Armstrong");  
    }  
}
```

10. Swap Numbers Without Temp

java

```
public class SwapNumbers {  
    public static void main(String[] args) {  
        int a = 5, b = 10;  
        a = a + b;  
        b = a - b;  
        a = a - b;  
        System.out.println("a=" + a + " b=" + b);  
    }  
}
```

```
}  
}
```

Category 3 – Arrays

11. Reverse Array

```
java  
  
import java.util.Arrays;  
public class ReverseArray {  
    public static void main(String[] args) {  
        int[] arr = {1,2,3,4};  
        for (int i = 0; i < arr.length/2; i++) {  
            int temp = arr[i];  
            arr[i] = arr[arr.length-1-i];  
            arr[arr.length-1-i] = temp;  
        }  
        System.out.println(Arrays.toString(arr));  
    }  
}
```

12. Largest & Smallest in Array

```
java  
  
public class MinMaxArray {  
    public static void main(String[] args) {  
        int[] arr = {4,2,9,1,5};  
        int min = arr[0], max = arr[0];  
        for (int n : arr) {  
            if (n < min) min = n;  
            if (n > max) max = n;  
        }  
        System.out.println("Min: " + min + ", Max: " + max);  
    }  
}
```

13. Second Largest in Array

java

```
public class SecondLargest {  
    public static void main(String[] args) {  
        int[] arr = {10, 20, 5, 8, 7};  
        int first = Integer.MIN_VALUE, second =  
Integer.MIN_VALUE;  
        for (int n : arr) {  
            if (n > first) {  
                second = first;  
                first = n;  
            } else if (n > second && n != first) {  
                second = n;  
            }  
        }  
        System.out.println("Second Largest: " + second);  
    }  
}
```

14. Sum of Array Elements

java

```
public class SumArray {  
    public static void main(String[] args) {  
        int[] arr = {1,2,3,4,5};  
        int sum = 0;  
        for (int n : arr) sum += n;  
        System.out.println("Sum: " + sum);  
    }  
}
```

15. Find Missing Number

java

```
public class MissingNumber {  
    public static void main(String[] args) {  
        int[] arr = {1,2,3,5};  
        int n = 5; // total numbers expected  
        int sum = n * (n+1) / 2;  
        int actualSum = 0;
```

```
        for (int num : arr) actualSum += num;
        System.out.println("Missing: " + (sum - actualSum));
    }
}
```

Category 4 – Collections

16. HashMap Iteration

```
java

import java.util.HashMap;
public class HashMapIterate {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("A", 1);
        map.put("B", 2);
        for (var entry : map.entrySet()) {
            System.out.println(entry.getKey() + "=" +
entry.getValue());
        }
    }
}
```

17. Sort List

```
java

import java.util.*;
public class SortList {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(5,2,9,1);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

18. Remove Duplicates from List

```
java

import java.util.*;
public class RemoveDupList {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1,2,2,3,4,4);
        Set<Integer> set = new LinkedHashSet<>(list);
        System.out.println(set);
    }
}
```

19. Frequency Count

```
java

import java.util.*;
public class FrequencyCount {
    public static void main(String[] args) {
        String str = "hello";
        Map<Character, Integer> freq = new HashMap<>();
        for (char c : str.toCharArray()) {
            freq.put(c, freq.getOrDefault(c, 0) + 1);
        }
        System.out.println(freq);
    }
}
```

20. Sort Map by Value

```
java

import java.util.*;
public class SortMapValue {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("A", 3); map.put("B", 1); map.put("C", 2);
        map.entrySet().stream()
            .sorted(Map.Entry.comparingByValue())
            .forEach(System.out::println);
    }
}
```


I can continue with **Category 5 to Category 10** including:

- **OOP Programs** (Singleton, Immutable, Builder, Factory Pattern)
- **File Handling** (Read/Write, BufferedReader, Serialization)
- **Multithreading** (Thread class, Runnable, Synchronization, Deadlock)
- **Java 8+ Features** (Streams, Lambdas, Optional)
- **Patterns & Advanced** (Observer, Strategy, Producer-Consumer)

If you want, I can give you **all remaining categories in one go** so you'll have a **complete 50+ program set** ready to practice.

Do you want me to continue and complete the kit in full?

Reverse a String, Check Palindrome, Anagram
Check don't use string and array default methods

Got it  — you want **pure logic implementations** without using **StringBuilder, reverse(), sort(), toCharArray()**, or other helper methods from `String / Arrays`.

We'll rely only on **loops and character operations**.

Here's the updated set:

1. Reverse a String (No Built-in Methods)

```
java

public class ReverseStringNoMethods {
    public static void main(String[] args) {
        String str = "Java";
        char[] chars = new char[str.length()]; // manual copy

        for (int i = 0; i < str.length(); i++) {
            chars[i] = str.charAt(i);
        }

        String reversed = "";
        for (int i = chars.length - 1; i >= 0; i--) {
            reversed += chars[i]; // concatenate manually
        }
    }
}
```

```

        System.out.println("Original: " + str);
        System.out.println("Reversed: " + reversed);
    }
}

```

Explanation:

- Created a char array manually using `charAt()`.
- Reversed by concatenating chars one by one (no reverse functions).

2. Check Palindrome (No Built-in Methods)

```

java

public class PalindromeNoMethods {
    public static void main(String[] args) {
        String str = "madam";
        boolean isPalindrome = true;

        for (int i = 0; i < str.length() / 2; i++) {
            if (str.charAt(i) != str.charAt(str.length() - 1 -
i)) {
                isPalindrome = false;
                break;
            }
        }

        System.out.println(str + " is " + (isPalindrome ?
"Palindrome" : "Not Palindrome"));
    }
}

```

Explanation:

- Compared first and last characters, moving towards the center.
- No string reversal, no helper methods.

3. Anagram Check (No Built-in Methods)

```

java

public class AnagramNoMethods {
    public static void main(String[] args) {
        String s1 = "listen";

```



```
String s2 = "silent";

if (s1.length() != s2.length()) {
    System.out.println("Not Anagram");
    return;
}

char[] arr1 = new char[s1.length()];
char[] arr2 = new char[s2.length()];

// Copy chars manually
for (int i = 0; i < s1.length(); i++) {
    arr1[i] = s1.charAt(i);
    arr2[i] = s2.charAt(i);
}

// Manual sorting (Bubble Sort)
for (int i = 0; i < arr1.length - 1; i++) {
    for (int j = i + 1; j < arr1.length; j++) {
        if (arr1[i] > arr1[j]) {
            char temp = arr1[i];
            arr1[i] = arr1[j];
            arr1[j] = temp;
        }
        if (arr2[i] > arr2[j]) {
            char temp = arr2[i];
            arr2[i] = arr2[j];
            arr2[j] = temp;
        }
    }
}

// Compare arrays manually
boolean isAnagram = true;
for (int i = 0; i < arr1.length; i++) {
    if (arr1[i] != arr2[i]) {
        isAnagram = false;
        break;
    }
}

System.out.println(isAnagram ? "Anagram" : "Not
Anagram");
}
```


Explanation:

- Converted to char arrays manually using `charAt()` .
 - Used **bubble sort** to sort arrays without built-in sort methods.
 - Compared each character one-by-one.
-

If you want, I can extend this "**No Built-in Methods**" set to include **Vowel Counting, Duplicate Removal, and String to Integer conversion** so you get the **full low-level logic pack** for interviews where built-in helpers are banned.

Do you want me to prepare that extended version next?

supposed to give example input and output for all Java coding master kit for interviews right?

Exactly  — a **Java Coding Master Kit** for interviews isn't really "interview-ready" unless **every program has an example input and output** so you can test quickly and know the expected results.

Right now, we have the first set of programs, but I can rewrite the **entire kit** so that:

- Each program includes **Example Input & Output** in comments.
 - You can **copy-paste and run** without guessing results.
 - Covers **core** → **advanced** programs.
-

Here's how I'll format it for you:

```
java

/*
Example Input:
Java
Example Output:
avaJ
*/

public class ReverseStringNoMethods {
```

```

public static void main(String[] args) {
    String str = "Java";
    char[] chars = new char[str.length()];
    for (int i = 0; i < str.length(); i++) {
        chars[i] = str.charAt(i);
    }
    String reversed = "";
    for (int i = chars.length - 1; i >= 0; i--) {
        reversed += chars[i];
    }
    System.out.println(reversed);
}
}

```

I can redo the **full 50+ program set** this way so you have:

- **Program** (clean & runnable)
- **Example Input & Output** (commented above the main class)
- **No confusion about expected result**

Do you want me to

1. **Redo the full kit** with this input/output format,
2. Or first give **just the String section** fully updated with examples before we do all 50+?

That way, you can confirm the style before I build the full master kit.

full 50+ program set

Got it — you want a **Java Coding Master Kit** with **50+ core interview programs**, each including:

- **Problem description**
- **Example input/output**
- **Java code implementation** (without over-relying on built-in helpers where possible)
- **Explanation** of logic

I can give you this in a **complete structured format** so it's like a "one-stop" coding interview reference.

Here's the **program list** I'll prepare for you:

Java Coding Master Kit — 50+ Essential Programs

String Programs

1. Reverse a String (no built-in reverse)
2. Check Palindrome (String)
3. Anagram Check (without Arrays.sort)
4. Count vowels and consonants
5. Remove duplicates from String
6. First non-repeating character in String
7. Find all permutations of String (Recursion)
8. Count occurrences of each character
9. Check if String contains only digits
10. Convert String to integer (without `Integer.parseInt`)

Number Programs

11. Reverse a number (no `String` methods)
12. Check Palindrome (Number)
13. Check Armstrong number
14. Check Prime number
15. Generate first N prime numbers
16. Fibonacci series (loop & recursion)
17. Factorial (loop & recursion)
18. Sum of digits of number
19. GCD and LCM of two numbers
20. Check perfect number

Array Programs

21. Reverse an array (no extra array)
22. Find largest and smallest element
23. Sort array (Bubble sort)
24. Find second largest number
25. Remove duplicates from array (no Set)
26. Merge two sorted arrays
27. Rotate array left/right
28. Find missing number in array (1 to N)
29. Find pair with given sum
30. Count frequency of elements

Pattern Printing

- 31. Pyramid pattern
- 32. Inverted pyramid
- 33. Diamond pattern
- 34. Floyd's triangle
- 35. Pascal's triangle

Matrix Programs

- 36. Transpose of a matrix
- 37. Matrix addition
- 38. Matrix multiplication
- 39. Check symmetric matrix
- 40. Rotate matrix 90 degrees

Data Structure Logic

- 41. Implement stack (array)
- 42. Implement queue (array)
- 43. Linked list basic insert/display
- 44. Reverse linked list
- 45. Detect cycle in linked list

Advanced

- 46. Singleton class (all variations)
- 47. Immutable class creation (all approaches)
- 48. Custom HashMap implementation (basic)
- 49. Implement binary search (recursive & iterative)
- 50. Implement linear search

If you want, I can now **start preparing the full kit** with:

- ✓ Example Input/Output
- ✓ Full code (no shortcuts for key logic)
- ✓ Step-by-step explanation

This will be a **ready-to-use Java interview handbook**.

Do you want me to start building it from **Program 1 to 50 in order** so you get a complete continuous reference? That way it will be like a clean PDF-ready version.

[start preparing the full kit](#)

Great — I'll deliver the **full Java Coding Master Kit (50 programs)**, ready to copy-paste into separate files and run. Each program includes:

- **Problem title**
- **Example input / output** (in comments)
- **Runnable Java program** (single-class, `main` method)
- **Short explanation**

I kept the **string problems** (reverse, palindrome, anagram) implemented **without using String/Array helper methods** (as you requested earlier). For other problems I use clear, straightforward logic (few utility methods where appropriate). Copy each class into its own `.java` file named after the class.

Java Coding Master Kit — 50 Programs

1. Reverse a String (no built-in reverse)

```
java

/*
Example Input:
Java
Example Output:
avaJ
*/

public class ReverseStringNoMethods {
    public static void main(String[] args) {
        String str = "Java";
        char[] chars = new char[str.length()];
        for (int i = 0; i < str.length(); i++) chars[i] =
str.charAt(i);
        String reversed = "";
        for (int i = chars.length - 1; i >= 0; i--) reversed
+= chars[i];
        System.out.println(reversed);
    }
}
```

Explanation: Manual char copy + manual concatenation in reverse order.

2. Check Palindrome (string, no built-ins)

```
java

/*
Input: madam
Output: madam is Palindrome
*/

public class PalindromeNoMethods {
    public static void main(String[] args) {
        String str = "madam";
        boolean isPalindrome = true;
        for (int i = 0; i < str.length() / 2; i++) {
            if (str.charAt(i) != str.charAt(str.length() - 1 - i)) {
                isPalindrome = false;
                break;
            }
        }
        System.out.println(str + " is " + (isPalindrome ? "Palindrome" : "Not Palindrome"));
    }
}
```

Explanation: Pairwise compare characters from ends.

3. Anagram Check (no Arrays.sort / no built-ins)

```
java

/*
Input: listen, silent
Output: Anagram
*/

public class AnagramNoMethods {
    public static void main(String[] args) {
        String s1 = "listen";
        String s2 = "silent";
        if (s1.length() != s2.length()) {
            System.out.println("Not Anagram");
        }
    }
}
```

```

        return;
    }
    char[] a = new char[s1.length()];
    char[] b = new char[s2.length()];
    for (int i = 0; i < s1.length(); i++) {
        a[i] = s1.charAt(i);
        b[i] = s2.charAt(i);
    }
    // simple bubble-like sort
    for (int i = 0; i < a.length - 1; i++) {
        for (int j = i + 1; j < a.length; j++) {
            if (a[i] > a[j]) { char t = a[i]; a[i] = a[j];
a[j] = t; }
            if (b[i] > b[j]) { char t = b[i]; b[i] = b[j];
b[j] = t; }
        }
    }
    boolean same = true;
    for (int i = 0; i < a.length; i++) if (a[i] != b[i]) {
same = false; break; }
    System.out.println(same ? "Anagram" : "Not Anagram");
}
}

```

Explanation: Manual char arrays + manual sort + compare.

4. Count Vowels & Consonants

```

java

/*
Input: Java Programming
Output: Vowels: 5, Consonants: 11
*/

public class VowelConsonantCount {
    public static void main(String[] args) {
        String s = "Java Programming";
        int vowels = 0, consonants = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = Character.toLowerCase(s.charAt(i));
            if (c >= 'a' && c <= 'z') {
                if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
vowels++; else consonants++;
            }
        }
    }
}

```



```

    }
    System.out.println("Vowels: " + vowels + ",
Consonants: " + consonants);
    }
}

```

Explanation: Loop and character checks.

5. Remove Duplicates from String (preserve order)

```

java

/*
Input: programming
Output: progamin
*/
public class RemoveDuplicatesString {
    public static void main(String[] args) {
        String s = "programming";
        String result = "";
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            boolean seen = false;
            for (int j = 0; j < result.length(); j++) if
(result.charAt(j) == c) { seen = true; break; }
            if (!seen) result += c;
        }
        System.out.println(result);
    }
}

```

Explanation: Build result string, skip chars already present.

6. First Non-Repeating Character

```

java

/*
Input: swiss
Output: w
*/
public class FirstNonRepeatingChar {
    public static void main(String[] args) {

```

```

String s = "swiss";
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    boolean unique = true;
    for (int j = 0; j < s.length(); j++) if (i != j &&
s.charAt(j) == c) { unique = false; break; }
    if (unique) { System.out.println(c); return; }
}
System.out.println("No non-repeating character");
}
}

```

Explanation: For each char check if it appears elsewhere.

7. All Permutations of a String (recursion)

```

java

/*
Input: ABC
Output: ABC ACB BAC BCA CAB CBA
*/

public class StringPermutations {
    public static void permute(String prefix, String s) {
        if (s.length() == 0) System.out.print(prefix + " ");
        else {
            for (int i = 0; i < s.length(); i++)
                permute(prefix + s.charAt(i), s.substring(0,
i) + s.substring(i + 1));
        }
    }
    public static void main(String[] args) {
        permute("", "ABC");
    }
}

```

Explanation: Standard recursion: pick char and permute rest.

8. Count Occurrences of Each Character

```

java

```

```

/*
Input: hello
Output: h:1 e:1 l:2 o:1
*/
public class CharFrequency {
    public static void main(String[] args) {
        String s = "hello";
        boolean[] printed = new boolean[s.length()];
        for (int i = 0; i < s.length(); i++) if (!printed[i])
        {
            char c = s.charAt(i);
            int count = 1;
            for (int j = i + 1; j < s.length(); j++) if
(s.charAt(j) == c) { count++; printed[j] = true; }
            System.out.print(c + ":" + count + " ");
        }
    }
}

```

Explanation: Mark duplicates to avoid double printing.

9. Check if String Contains Only Digits

```

java

/*
Input: 12345
Output: true
*/
public class IsNumericString {
    public static void main(String[] args) {
        String s = "12345";
        boolean numeric = s.length() > 0;
        for (int i = 0; i < s.length() && numeric; i++) {
            char c = s.charAt(i);
            if (c < '0' || c > '9') numeric = false;
        }
        System.out.println(numeric);
    }
}

```

Explanation: Check each char for digit range.

10. Convert String to Integer (no Integer.parseInt)

```
java

/*
Input: "1234"
Output: 1234
*/

public class StringToInteger {
    public static void main(String[] args) {
        String s = "1234";
        int sign = 1, i = 0;
        if (s.charAt(0) == '-') { sign = -1; i = 1; }
        int num = 0;
        for (; i < s.length(); i++) num = num * 10 +
(s.charAt(i) - '0');
        System.out.println(num * sign);
    }
}
```

Explanation: Build number digit-by-digit.

11. Reverse a Number (no String methods)

```
java

/*
Input: 1234
Output: 4321
*/

public class ReverseNumber {
    public static void main(String[] args) {
        int n = 1234;
        int rev = 0;
        while (n != 0) {
            rev = rev * 10 + n % 10;
            n /= 10;
        }
        System.out.println(rev);
    }
}
```

Explanation: Use modulo/division.

12. Check Number Palindrome

```
java

/*
Input: 121
Output: Palindrome
*/

public class NumberPalindrome {
    public static void main(String[] args) {
        int n = 121, original = n, rev = 0;
        while (n != 0) { rev = rev * 10 + n % 10; n /= 10; }
        System.out.println(rev == original ? "Palindrome" :
"Not Palindrome");
    }
}
```

Explanation: Reverse numerically and compare.

13. Armstrong Number (n-digit)

```
java

/*
Input: 153
Output: Armstrong
*/

public class ArmstrongNumber {
    public static void main(String[] args) {
        int n = 153, temp = n, sum = 0;
        int digits = 0, t = n;
        while (t != 0) { digits++; t /= 10; }
        while (temp != 0) {
            int d = temp % 10;
            int p = 1;
            for (int i = 0; i < digits; i++) p *= d;
            sum += p;
            temp /= 10;
        }
        System.out.println(sum == n ? "Armstrong" : "Not
Armstrong");
    }
}
```

Explanation: Raise each digit to number-of-digits power.

14. Check Prime Number

```
java

/*
Input: 29
Output: Prime
*/
public class PrimeCheck {
    public static void main(String[] args) {
        int n = 29;
        if (n <= 1) { System.out.println("Not Prime"); return; }

        boolean prime = true;
        for (int i = 2; i * i <= n; i++) if (n % i == 0) {
prime = false; break; }
        System.out.println(prime ? "Prime" : "Not Prime");
    }
}
```

Explanation: Test divisors until \sqrt{n} .

15. Generate First N Prime Numbers

```
java

/*
Input: n=10
Output: 2 3 5 7 11 13 17 19 23 29
*/
public class FirstNPrimes {
    public static void main(String[] args) {
        int n = 10, count = 0, num = 2;
        while (count < n) {
            boolean prime = true;
            for (int i = 2; i * i <= num; i++) if (num % i ==
0) { prime = false; break; }
            if (prime) { System.out.print(num + " "); count++; }

            num++;
        }
    }
}
```

```
    }
}
```

Explanation: Simple trial division.

16. Fibonacci Series (loop)

```
java

/*
Input: n=6
Output: 0 1 1 2 3 5
*/
public class FibonacciLoop {
    public static void main(String[] args) {
        int n = 6;
        int a = 0, b = 1;
        if (n > 0) System.out.print(a);
        if (n > 1) System.out.print(" " + b);
        for (int i = 2; i < n; i++) {
            int c = a + b; System.out.print(" " + c);
            a = b; b = c;
        }
    }
}
```

Explanation: Iteratively produce sequence.

17. Factorial (recursion & loop)

```
java

/*
Input: 5
Output (recursive): 120
Output (iterative): 120
*/
public class FactorialBoth {
    static long factRec(int n) { return (n<=1)?1:n*factRec(n-1); }
    static long factIter(int n) { long r=1; for(int i=2;i<=n;i++) r*=i; return r; }
    public static void main(String[] args) {
```

```

        int n = 5;
        System.out.println("Recursive: " + factRec(n));
        System.out.println("Iterative: " + factIter(n));
    }
}

```

Explanation: Two approaches for practice.

18. Sum of Digits

```

java

/*
Input: 1234
Output: 10
*/
public class SumOfDigits {
    public static void main(String[] args) {
        int n = 1234, sum = 0;
        while (n != 0) { sum += n % 10; n /= 10; }
        System.out.println(sum);
    }
}

```

Explanation: Use modulo and division.

19. GCD and LCM

```

java

/*
Input: 12, 18
Output: GCD=6, LCM=36
*/
public class GcdLcm {
    static int gcd(int a, int b) { while (b != 0) { int t = b;
b = a % b; a = t; } return a; }
    public static void main(String[] args) {
        int a = 12, b = 18;
        int g = gcd(a, b);
        int l = (a / g) * b;
        System.out.println("GCD=" + g + ", LCM=" + l);
    }
}

```



```

    }
}

```

Explanation: Euclidean algorithm.

20. Check Perfect Number

```

java

/*
Input: 28
Output: Perfect Number
*/
public class PerfectNumber {
    public static void main(String[] args) {
        int n = 28;
        int sum = 1;
        for (int i = 2; i * i <= n; i++) {
            if (n % i == 0) {
                sum += i;
                if (i != n / i) sum += n / i;
            }
        }
        if (n == 1) sum = 0;
        System.out.println(sum == n ? "Perfect Number" : "Not
Perfect");
    }
}

```

Explanation: Sum of proper divisors.

21. Reverse an Array (in-place)

```

java

/*
Input: [1,2,3,4]
Output: [4,3,2,1]
*/
import java.util.Arrays;
public class ReverseArray {
    public static void main(String[] args) {
        int[] a = {1,2,3,4};
    }
}

```

```

        for (int i = 0; i < a.length/2; i++) {
            int t = a[i]; a[i] = a[a.length-1-i]; a[a.length-1-i] = t;
        }
        System.out.println(Arrays.toString(a));
    }
}

```

Explanation: Swap symmetric elements.

22. Find Largest & Smallest in Array

```

java

/*
Input: [4,2,9,1,5]
Output: Min:1, Max:9
*/
public class MinMaxArray {
    public static void main(String[] args) {
        int[] a = {4,2,9,1,5};
        int min = a[0], max = a[0];
        for (int v : a) { if (v < min) min = v; if (v > max)
max = v; }
        System.out.println("Min:" + min + ", Max:" + max);
    }
}

```

Explanation: Single-pass scanning.

23. Bubble Sort

```

java

/*
Input: [5,2,9,1]
Output: [1,2,5,9]
*/
import java.util.Arrays;
public class BubbleSort {
    public static void main(String[] args) {
        int[] a = {5,2,9,1};
        for (int i = 0; i < a.length - 1; i++)

```

```

        for (int j = 0; j < a.length - 1 - i; j++)
            if (a[j] > a[j+1]) { int t = a[j]; a[j] =
a[j+1]; a[j+1] = t; }
        System.out.println(Arrays.toString(a));
    }
}

```

Explanation: Classic bubble pass swaps.

24. Second Largest Number

```

java

/*
Input: [10,20,5,8,7]
Output: 10 (second Largest)
*/

public class SecondLargest {
    public static void main(String[] args) {
        int[] a = {10,20,5,8,7};
        int first = Integer.MIN_VALUE, second =
Integer.MIN_VALUE;
        for (int v : a) {
            if (v > first) { second = first; first = v; }
            else if (v > second && v != first) second = v;
        }
        System.out.println("Second Largest: " + second);
    }
}

```

Explanation: Track first & second in one pass.

25. Remove Duplicates from Array (no Set)

```

java

/*
Input: [1,2,2,3,4,4]
Output: [1,2,3,4]
*/

import java.util.Arrays;
public class RemoveDupArray {
    public static void main(String[] args) {

```

```

int[] a = {1,2,2,3,4,4};
int n = a.length;
Arrays.sort(a); // allowed here for simplicity
int index = 0;
for (int i = 0; i < n; i++) if (i == 0 || a[i] != a[i-1]) a[index++] = a[i];
int[] res = new int[index];
System.arraycopy(a, 0, res, 0, index);
System.out.println(Arrays.toString(res));
}
}

```

Explanation: Sort then compress duplicates.

26. Merge Two Sorted Arrays

```

java

/*
Input: [1,3,5] and [2,4,6]
Output: [1,2,3,4,5,6]
*/
import java.util.Arrays;
public class MergeSortedArrays {
    public static void main(String[] args) {
        int[] a = {1,3,5}, b = {2,4,6};
        int[] c = new int[a.length + b.length];
        int i=0,j=0,k=0;
        while (i<a.length && j<b.length) c[k++] = (a[i]
<=b[j])?a[i++]:b[j++];
        while (i<a.length) c[k++] = a[i++];
        while (j<b.length) c[k++] = b[j++];
        System.out.println(Arrays.toString(c));
    }
}

```

Explanation: Two-pointer merge.

27. Rotate Array Left by d (in-place using reversal)

```

java

```

```

/*
Input: [1,2,3,4,5], d=2
Output: [3,4,5,1,2]
*/
import java.util.Arrays;
public class RotateArrayLeft {
    static void reverse(int[] a, int l, int r) { while (l<r) {
int t=a[l]; a[l++]=a[r]; a[r--]=t; } }
    public static void main(String[] args) {
        int[] a = {1,2,3,4,5}; int d = 2; d %= a.length;
        reverse(a, 0, d-1); reverse(a, d, a.length-1);
reverse(a, 0, a.length-1);
        System.out.println(Arrays.toString(a));
    }
}

```

Explanation: Use triple reversal trick.

28. Find Missing Number in 1..N

```

java

/*
Input: [1,2,4,5], N=5
Output: 3
*/
public class MissingNumber {
    public static void main(String[] args) {
        int[] a = {1,2,4,5};
        int n = 5;
        int sum = n * (n+1) / 2;
        int s = 0; for (int v : a) s += v;
        System.out.println(sum - s);
    }
}

```

Explanation: Sum formula.

29. Find Pair with Given Sum (sorted input)

```

java

```

```

/*
Input: [1,2,3,4,5], target=9
Output: Pair found: 4,5
*/
public class PairWithSum {
    public static void main(String[] args) {
        int[] a = {1,2,3,4,5}; int target = 9;
        int l = 0, r = a.length - 1;
        while (l < r) {
            int s = a[l] + a[r];
            if (s == target) { System.out.println("Pair found:
" + a[l] + "," + a[r]); return; }
            else if (s < target) l++; else r--;
        }
        System.out.println("No pair found");
    }
}

```

Explanation: Two-pointer approach.

30. Count Frequency of Elements (array)

```

java

/*
Input: [1,2,2,3]
Output: 1:1 2:2 3:1
*/
public class FrequencyArray {
    public static void main(String[] args) {
        int[] a = {1,2,2,3};
        boolean[] seen = new boolean[a.length];
        for (int i = 0; i < a.length; i++) if (!seen[i]) {
            int count = 1;
            for (int j = i+1; j < a.length; j++) if (a[i] ==
a[j]) { count++; seen[j] = true; }
            System.out.print(a[i] + ":" + count + " ");
        }
    }
}

```

Explanation: Quadratic approach good for small arrays.

31. Pyramid Pattern

```

java

/*
Input: n=4
Output:
    *
   ***
  *****
 */

public class PyramidPattern {
    public static void main(String[] args) {
        int n = 4;
        for (int i = 1; i <= n; i++) {
            for (int s = 0; s < n - i; s++) System.out.print(" ");
            for (int k = 0; k < (2*i - 1); k++)
                System.out.print("*");
            System.out.println();
        }
    }
}

```

Explanation: Print spaces then stars.

32. Inverted Pyramid

```

java

/*
Input: n=4
Output:
*****
****
***
**
*
*/

public class InvertedPyramid {
    public static void main(String[] args) {
        int n = 4;
        for (int i = n; i >= 1; i--) {
            for (int s = 0; s < n - i; s++) System.out.print(" ");

```

```
");
        for (int k = 0; k < (2*i - 1); k++)
            System.out.print("*");
        System.out.println();
    }
}
```

Explanation: Reverse of pyramid.

33. Diamond Pattern

```
java

/*
Input: n=3
Output:
  *
 ***
*****
 ***
  *
*/

public class DiamondPattern {
    public static void main(String[] args) {
        int n = 3;
        for (int i = 1; i <= n; i++) {
            for (int s=0; s<n-i; s++) System.out.print(" ");
            for (int k=0; k<2*i-1; k++) System.out.print("*");
            System.out.println();
        }
        for (int i = n-1; i >= 1; i--) {
            for (int s=0; s<n-i; s++) System.out.print(" ");
            for (int k=0; k<2*i-1; k++) System.out.print("*");
            System.out.println();
        }
    }
}
```

Explanation: Upper pyramid + lower pyramid.

34. Floyd's Triangle


```

java

/*
Input: n=4
Output:
1
2 3
4 5 6
7 8 9 10
*/

public class FloydTriangle {
    public static void main(String[] args) {
        int n = 4, num = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++)
                System.out.print((num++) + " ");
            System.out.println();
        }
    }
}

```

Explanation: Incremental numbers per row.

35. Pascal's Triangle

```

java

/*
Input: n=5
Output:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
*/

public class PascalsTriangle {
    public static void main(String[] args) {
        int n = 5;
        for (int i = 0; i < n; i++) {
            int val = 1;
            for (int j = 0; j <= i; j++) {
                System.out.print(val + " ");
                val = val * (i - j) / (j + 1);
            }
        }
    }
}

```

```

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

```

Explanation: Use combinatorial relation.

36. Transpose of a Matrix

```

java

/*
Input:
1 2
3 4
Output:
1 3
2 4
*/

public class TransposeMatrix {
    public static void main(String[] args) {
        int[][] m = {{1,2},{3,4}};
        int r = m.length, c = m[0].length;
        int[][] t = new int[c][r];
        for (int i=0;i<r;i++) for (int j=0;j<c;j++) t[j][i] =
m[i][j];
        for (int i=0;i<t.length;i++) {
            for (int j=0;j<t[0].length;j++)
                System.out.print(t[i][j] + " ");
            System.out.println();
        }
    }
}

```

Explanation: Swap indexes.

37. Matrix Addition

```

java

/*
Input: [[1,2],[3,4]] + [[5,6],[7,8]]

```

```

Output: [[6,8],[10,12]]
*/
public class MatrixAddition {
    public static void main(String[] args) {
        int[][] a = {{1,2},{3,4}}, b = {{5,6},{7,8}};
        int r=a.length, c=a[0].length;
        int[][] s = new int[r][c];
        for (int i=0;i<r;i++) for (int j=0;j<c;j++) s[i]
[j]=a[i][j]+b[i][j];
        for (int i=0;i<r;i++){ for(int j=0;j<c;j++)
System.out.print(s[i][j]+" "); System.out.println(); }
    }
}

```

Explanation: Element-wise addition.

38. Matrix Multiplication

```

java

/*
Input: A=[[1,2],[3,4]] B=[[5,6],[7,8]]
Output: [[19,22],[43,50]]
*/
public class MatrixMultiplication {
    public static void main(String[] args) {
        int[][] A = {{1,2},{3,4}}, B = {{5,6},{7,8}};
        int r = A.length, m = B.length, c = B[0].length;
        int[][] C = new int[r][c];
        for (int i=0;i<r;i++)
            for (int j=0;j<c;j++)
                for (int k=0;k<m;k++) C[i][j] += A[i][k]*B[k]
[j];
        for (int i=0;i<r;i++){ for (int j=0;j<c;j++)
System.out.print(C[i][j]+" "); System.out.println(); }
    }
}

```

Explanation: Triple nested loops.

39. Check Symmetric Matrix

```

java

```

```

/*
Input: [[1,2],[2,1]]
Output: Symmetric
*/
public class SymmetricMatrix {
    public static void main(String[] args) {
        int[][] m = {{1,2},{2,1}};
        boolean sym = true;
        for (int i=0;i<m.length;i++)
            for (int j=i+1;j<m[0].length;j++)
                if (m[i][j] != m[j][i]) sym = false;
        System.out.println(sym ? "Symmetric" : "Not
Symmetric");
    }
}

```

Explanation: Compare $m[i][j]$ with $m[j][i]$.

40. Rotate Matrix 90 Degrees (clockwise)

```

java

/*
Input:
1 2
3 4
Output:
3 1
4 2
*/
import java.util.Arrays;
public class RotateMatrix90 {
    public static void main(String[] args) {
        int[][] a = {{1,2},{3,4}};
        int n = a.length;
        // transpose
        for (int i=0;i<n;i++) for (int j=i+1;j<n;j++) { int
t=a[i][j]; a[i][j]=a[j][i]; a[j][i]=t; }
        // reverse each row
        for (int i=0;i<n;i++) {
            for (int l=0, r=n-1; l<r; l++, r--) { int t=a[i]
[l]; a[i][l]=a[i][r]; a[i][r]=t; }
        }
    }
}

```

```

        for (int i=0;i<n;i++)
System.out.println(Arrays.toString(a[i]));
    }
}

```

Explanation: Transpose then reverse rows.

41. Implement Stack using Array

```

java

/*
Operations demonstration
*/
import java.util.EmptyStackException;
public class ArrayStack {
    private int[] arr; private int top = -1;
    public ArrayStack(int capacity) { arr=new int[capacity]; }
    public void push(int x) { if (top==arr.length-1) throw new
RuntimeException("Overflow"); arr[++top]=x; }
    public int pop() { if (top== -1) throw new
EmptyStackException(); return arr[top--]; }
    public int peek() { if (top== -1) throw new
EmptyStackException(); return arr[top]; }
    public boolean isEmpty() { return top== -1; }
    public static void main(String[] args) {
        ArrayStack s = new ArrayStack(5);
        s.push(10); s.push(20); System.out.println(s.pop());
        System.out.println(s.peek());
    }
}

```

Explanation: Basic stack operations.

42. Implement Queue using Array (circular)

```

java

/*
Demonstration of enqueue/dequeue
*/
public class ArrayQueue {
    private int[] arr; private int head=0, tail=0, size=0;

```

```

    public ArrayQueue(int cap){ arr=new int[cap]; }
    public void enqueue(int x){ if(size==arr.length) throw new
RuntimeException("Full"); arr[tail]=x; tail=
(tail+1)%arr.length; size++; }
    public int dequeue(){ if(size==0) throw new
RuntimeException("Empty"); int v=arr[head]; head=
(head+1)%arr.length; size--; return v; }
    public boolean isEmpty(){ return size==0; }
    public static void main(String[] args){
        ArrayQueue q = new ArrayQueue(5); q.enqueue(1);
q.enqueue(2); System.out.println(q.dequeue());
System.out.println(q.dequeue());
    }
}

```

Explanation: Circular buffer.

43. Singly Linked List — insert & display

```

java

/*
Insert 1,2,3 -> display: 1 2 3
*/
public class LinkedListBasic {
    static class Node { int val; Node next; Node(int v)
{val=v;} }
    Node head;
    void insert(int v){ Node n=new Node(v); if(head==null)
head=n; else{ Node t=head; while(t.next!=null) t=t.next;
t.next=n; } }
    void display(){ Node t=head; while(t!=null){
System.out.print(t.val+" "); t=t.next; } System.out.println();
}
    public static void main(String[] args){
        LinkedListBasic list=new LinkedListBasic();
        list.insert(1); list.insert(2); list.insert(3);
list.display();
    }
}

```

Explanation: Simple tail insertion.

44. Reverse a Linked List (iterative)

```

java

/*
Input: 1->2->3
Output: 3->2->1
*/

public class ReverseLinkedList {
    static class Node { int val; Node next; Node(int v)
    {val=v;} }
    static Node reverse(Node head) {
        Node prev = null, curr = head;
        while (curr != null) { Node next = curr.next;
        curr.next = prev; prev = curr; curr = next; }
        return prev;
    }
    public static void main(String[] args) {
        Node a = new Node(1); a.next = new Node(2);
        a.next.next = new Node(3);
        Node r = reverse(a);
        while (r != null) { System.out.print(r.val + " "); r =
        r.next; }
    }
}

```

Explanation: Classic iterative reversal.

45. Detect Cycle in Linked List (Floyd's)

```

java

/*
Create cycle -> detect true
*/

public class DetectCycle {
    static class Node { int val; Node next; Node(int v)
    {val=v;} }
    static boolean hasCycle(Node head) {
        Node slow=head, fast=head;
        while (fast!=null && fast.next!=null) {
        slow=slow.next; fast=fast.next.next; if (slow==fast) return
        true; }
        return false;
    }
}

```

```

    }
    public static void main(String[] args) {
        Node a = new Node(1); Node b=new Node(2); Node c=new
Node(3);
        a.next=b; b.next=c; c.next=b; // cycle
        System.out.println(hasCycle(a));
    }
}

```

Explanation: Two-pointer meeting indicates cycle.

46. Singleton Patterns — examples (Eager, Lazy, Double-Checked, Bill Pugh, Enum)

```

java

/*
Usage demonstration for different singleton types is embedded
as main methods in each class below.
*/

// Due to space, here is Bill Pugh and Enum – others already
shown previously in kit.
public class BillPughSingleton {
    private BillPughSingleton() {}
    private static class Helper { private static final
BillPughSingleton INST = new BillPughSingleton(); }
    public static BillPughSingleton getInstance() { return
Helper.INST; }
    public static void main(String[] args) {
        System.out.println(BillPughSingleton.getInstance() ==
BillPughSingleton.getInstance()); }
}
enum EnumSingleton { INSTANCE; public void demo(){
    System.out.println("Enum"); } }

```

Explanation: Use inner helper or enum for safety.

47. Immutable Class Examples

```

java

/*
Immutable Person example

```



```

*/
public final class ImmutablePerson {
    private final String name; private final int age;
    public ImmutablePerson(String name, int age) { this.name =
name; this.age = age; }
    public String getName(){ return name; } public int
getAge(){ return age; }
    public static void main(String[] args){ ImmutablePerson p
= new ImmutablePerson("Alice",30);
System.out.println(p.getName()+" "+p.getAge()); }
}

```

Explanation: final class, private final fields, no setters.

48. Basic Custom HashMap (very simple, not production-ready)

```

java

/*
 * Demonstration of put/get using separate chaining
 */
import java.util.ArrayList;
import java.util.List;
public class SimpleHashMap {
    static class Entry { String key; int val; Entry(String
k,int v){key=k;val=v;} }
    List<Entry>[] table;
    int capacity;
    @SuppressWarnings("unchecked")
    public SimpleHashMap(int cap) { capacity = cap; table =
(List<Entry>[]) new List[cap]; for(int i=0;i<cap;i++)
table[i]=new ArrayList<>(); }
    void put(String k, int v) {
        int idx = Math.abs(k.hashCode()) % capacity;
        for (Entry e : table[idx]) if (e.key.equals(k)) {
e.val = v; return; }
        table[idx].add(new Entry(k,v));
    }
    Integer get(String k) {
        int idx = Math.abs(k.hashCode()) % capacity;
        for (Entry e : table[idx]) if (e.key.equals(k)) return
e.val;
        return null;
    }
}

```

```

    }
    public static void main(String[] args) {
        SimpleHashMap m = new SimpleHashMap(10); m.put("A",1);
        m.put("B",2); System.out.println(m.get("A"));
    }
}

```

Explanation: Very simplified bucket map.

49. Binary Search (iterative & recursive)

```

java

/*
Input: [1,2,3,4,5], target=3 -> index 2
*/
public class BinarySearch {
    static int iter(int[] a, int target) {
        int l=0,r=a.length-1;
        while (l<=r) {
            int mid = l + (r-l)/2;
            if (a[mid]==target) return mid;
            else if (a[mid]<target) l=mid+1; else r=mid-1;
        }
        return -1;
    }
    static int rec(int[] a, int l, int r, int target) {
        if (l>r) return -1;
        int mid = l + (r-l)/2;
        if (a[mid]==target) return mid;
        return a[mid] > target ? rec(a, l, mid-1, target) :
        rec(a, mid+1, r, target);
    }
    public static void main(String[] args) {
        int[] a = {1,2,3,4,5}; System.out.println(iter(a,3));
        System.out.println(rec(a,0,a.length-1,3));
    }
}

```

Explanation: Standard binary search.

50. Linear Search

```
java

/*
Input: [4,2,9,1], target=9 -> index 2
*/
public class LinearSearch {
    public static void main(String[] args) {
        int[] a = {4,2,9,1}; int target = 9;
        for (int i=0;i<a.length;i++) if (a[i]==target) {
            System.out.println(i); return; }
        System.out.println(-1);
    }
}
```

Explanation: Simple scan.

Notes & Next Steps

- This kit contains **50 essential programs** spanning Strings, Numbers, Arrays, Patterns, Matrices, basic Data Structures, and core Algorithms.
- **Each class is self-contained** — save as `ClassName.java` and run in your IDE / `javac` + `java` .
- For large programs (e.g., HashMap), this is a **simple educational implementation**, not production-ready — good for interviews and learning internals.

If you want, I can now:

1. **Produce these as a downloadable ZIP** with files (I can generate code files in-line if you want them packaged — tell me you want files).
2. **Extend the kit** with:
 - Multithreading examples (synchronized, volatile, wait/notify, thread-pools)
 - Java 8+ features (streams, lambdas, Optional)
 - Design patterns (Factory, Strategy, Observer) with code
 - More advanced DS (binary tree, BST operations, heap, graph DFS/BFS)

Which of the above would you like next?

what is Anagram and amstrom number

Let's break them down clearly.

1. Anagram

- **Definition:**

Two strings are **Anagrams** if they contain the **same characters** with the **same frequency**, but possibly in a different order.

Spaces and case are often ignored in checks (depends on the problem statement).

- **Example:**

pgsql

Input: str1 = "listen", str2 = "silent"

Output: true

Explanation: Both contain 'l', 'i', 's', 't', 'e', 'n' exactly once.

- **Non-Anagram Example:**

vbnet

Input: str1 = "hello", str2 = "bello"

Output: false

Explanation: Characters differ.

2. Armstrong Number

- **Definition:**

A number is an **Armstrong Number** (also called a Narcissistic Number) if the **sum of its own digits each raised to the power of the number of digits** equals the number itself.

- **Example for 3-digit Armstrong Number:**

vbnet

Input: 153

Calculation: $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$

Output: true

- **Example for 4-digit Armstrong Number:**

yarn

Input: 9474

Calculation: $9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + 2401 + 256 = 9474$

Output: true

- **Non-Armstrong Example:**

vbnet

Input: 123

Calculation: $1^3 + 2^3 + 3^3 = 36 \neq 123$

Output: false

If you want, I can now prepare **Java programs for both Anagram check and Armstrong number check** with:

- No default methods like `Arrays.sort()` or `String.toCharArray()`
- With **example input/output** so we can start adding them to your **Java Coding Master Kit**.

Do you want me to add them as the **first 2 programs** in your 50+ set?

give example for read values from set

Here's a **simple Java example** for reading values from a `Set` and printing them:

Example: Reading values from a `Set`

java

```
import java.util.*;

public class ReadSetExample {
    public static void main(String[] args) {
        // Create a Set and add some values
        Set<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
    }
}
```

```
fruits.add("Orange");

// Example 1: Using for-each loop
System.out.println("Using for-each loop:");
for (String fruit : fruits) {
    System.out.println(fruit);
}

// Example 2: Using Iterator
System.out.println("\nUsing Iterator:");
Iterator<String> iterator = fruits.iterator();
while (iterator.hasNext()) {
    String fruit = iterator.next();
    System.out.println(fruit);
}

// Example 3: Using Java 8 forEach and Lambda
System.out.println("\nUsing forEach and Lambda:");
fruits.forEach(fruit -> System.out.println(fruit));
}
}
```

Example Output

(Order may vary because `HashSet` does not maintain insertion order)

vbnet

Using for-each loop:

Banana

Apple

Orange

Mango

Using Iterator:

Banana

Apple

Orange

Mango

Using forEach and Lambda:

Banana

Apple

Orange
Mango

If you want, I can also give you **examples of reading values from a `Set` while maintaining insertion order** using `LinkedHashSet` or sorting them with `TreeSet`. That will be useful in your **Java coding master kit**.

is get not possible with set?

Exactly — in Java, a `Set` does **not** have a `get(index)` method like a `List` does, because:

- `Set` is designed for **uniqueness** of elements, not for **positional access**.
- The elements in a `Set` are **unordered** (except for ordered implementations like `LinkedHashSet`), so the concept of an index doesn't apply.

If you want to *read* values from a `Set`, you typically:

1 Using Iterator

```
java

import java.util.*;

public class ReadSetExample {
    public static void main(String[] args) {
        Set<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        Iterator<String> iterator = fruits.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Output:

nginx

```
Apple  
Banana  
Cherry
```

(Order may vary because `HashSet` is unordered.)

2 Using Enhanced for loop

```
java  
  
for (String fruit : fruits) {  
    System.out.println(fruit);  
}
```

3 Convert to List for indexed access

```
java  
  
List<String> fruitList = new ArrayList<>(fruits);  
System.out.println(fruitList.get(1)); // Now you can use get()
```

✓ Key takeaway:

- `Set` → no `get(index)`
- If you need **index-based** access → convert to a `List`
- If you just need to read all elements → use `for-each` or `Iterator` .

If you want, I can prepare a **full table of Set reading techniques with examples and outputs** so it's part of your *Java Coding Master Kit*. That way it's ready for interview prep.

i need to get exact value from set, how we can do

In Java, a `Set` doesn't have a `get(index)` method like a `List` because it is **unordered** and doesn't store elements by position.

But you **can** get an exact value if you know what you're looking for — you just can't do it by index.

1 Check if a value exists and retrieve it

If you already know the object you want, you can use `contains()` to check and then iterate to get it:

```
java

import java.util.HashSet;
import java.util.Set;

public class GetValueFromSet {
    public static void main(String[] args) {
        Set<String> names = new HashSet<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        String search = "Bob";
        String found = null;

        for (String name : names) {
            if (name.equals(search)) {
                found = name;
                break;
            }
        }

        if (found != null) {
            System.out.println("Found: " + found);
        } else {
            System.out.println("Not Found");
        }
    }
}
```

Output:

```
makefile

Found: Bob
```

2 Get element by position (convert to List)

If you really want to get a value **by index** from a `Set`, you must convert it to a `List` first.

```
java

import java.util.*;

public class SetToListExample {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<>(); // Keeps
        insertion order
        set.add("Red");
        set.add("Green");
        set.add("Blue");

        List<String> list = new ArrayList<>(set);

        String secondElement = list.get(1); // Index 1 =
        "Green"
        System.out.println("Second element: " +
        secondElement);
    }
}
```

Output:

```
sql

Second element: Green
```

3 When to use which

- If you know the value → Use `contains()` or loop.
- If you need index access → Use a `List` instead of a `Set` OR convert `Set` to `List`.
- If you need both fast lookup & index access → Use a `LinkedHashSet` (for order) and convert when needed.

If you want, I can add a “Get element from Set” section to your Java Coding Master Kit so you remember all 3 ways.

Do you want me to do that while preparing the kit?

give example for Van Eck sequence with complete code

Got it — you want a **full working Java program** to generate the **Van Eck sequence**.

Let's first understand what it is, then I'll give you the **complete code with example input and output**.

What is the Van Eck sequence?

- A sequence of integers where:
 1. Start with `0`.
 2. For each next term:
 - If the **previous term** has not appeared before, append `0`.
 - If it **has appeared** before, append the **distance to its previous occurrence**.
- It's defined in OEIS as [A181391](#).

Example:

makefile

```
Index:  0  1  2  3  4  5  6  7  8  9 10 11 ...
Value:  0, 0, 1, 0, 2, 0, 2, 2, 1, 6, 0, 5, ...
```

Java Code for Van Eck Sequence

java

```
import java.util.*;

public class VanEckSequence {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of terms: ");
        int n = sc.nextInt();

        List<Integer> sequence = new ArrayList<>();
```

```

Map<Integer, Integer> lastSeen = new HashMap<>();

// First term is always 0
sequence.add(0);
lastSeen.put(0, 0);

for (int i = 1; i < n; i++) {
    int prev = sequence.get(i - 1);

    if (lastSeen.containsKey(prev) &&
lastSeen.get(prev) != i - 1) {
        int lastIndex = lastSeen.get(prev);
        sequence.add((i - 1) - lastIndex);
    } else {
        sequence.add(0);
    }

    // Update last seen position of previous number
    lastSeen.put(prev, i - 1);
}

// Print sequence
System.out.println("Van Eck sequence (" + n + "
terms): " + sequence);
}
}

```

Example Run

Input:

```
typescript
```

```
Enter number of terms: 15
```

Output:

```
java
```

```
Van Eck sequence (15 terms): [0, 0, 1, 0, 2, 0, 2, 2, 1, 6, 0,
5, 0, 2, 6]
```

Explanation of Steps

1. Start with 0 .
2. Keep a map to store the last seen index of each number.
3. For each new term:
 - If the last term was seen before (not counting the immediate last position), take the distance between occurrences.
 - Otherwise, append 0 .
4. Update the map with the last occurrence position.

If you want, I can also make a **Van Eck sequence generator** that

⚡ works in $O(n)$ time and memory,

⚡ and directly prints values **without storing full history** for memory efficiency.

Do you want me to prepare that optimized version?

take 10 terms and explain step by step what happens

Van Eck sequence — first 10 terms (step-by-step)

Sequence rule (quick):

Start with $a_0 = 0$. For each $n \geq 1$ let $k = a_{n-1}$ (the previous term).

- If k has not appeared before index $n-1$, then $a[n] = 0$.
- If k has appeared before at index $j < n-1$, then $a[n] = (n-1) - j$ (the distance to its last prior occurrence).

Below are the first 10 terms (indices $0..9$) and a step-by-step explanation of how each term is produced.

First 10 terms:

$[0, 0, 1, 0, 2, 0, 2, 2, 1, 6]$

Step-by-step

Index 0

- $a_0 = 0$ (by definition).
- Nothing to compute.

Index 1

- $prev = a_0 = 0$.
- Does 0 occur *before* index 0 ? → **No**.
- So $a_1 = 0$.
Calculation: no prior occurrence $\Rightarrow a_1 = 0$.

Index 2

- $prev = a_1 = 0$.
- Does 0 occur *before* index 1 ? → **Yes**, at index 0 . ($j = 0$)
- So $a_2 = (1) - (0) = 1$.
Calculation: $(n-1) = 1$, $j = 0 \Rightarrow 1 - 0 = 1$.

Index 3

- $prev = a_2 = 1$.
- Does 1 occur *before* index 2 ? → **No** (first time at index 2).
- So $a_3 = 0$.
Calculation: no prior occurrence $\Rightarrow a_3 = 0$.

Index 4

- $prev = a_3 = 0$.
- Where did 0 appear before index 3 ? Occurrences at indices 0 and 1 (and 3 is the current prev). The **last prior** occurrence is $j = 1$.
- So $a_4 = (3) - (1) = 2$.
Calculation: $(n-1) = 3$, $j = 1 \Rightarrow 3 - 1 = 2$.

Index 5

- $prev = a_4 = 2$.
- Does 2 occur *before* index 4 ? → **No** (first time at index 4).
- So $a_5 = 0$.
Calculation: no prior occurrence $\Rightarrow a_5 = 0$.

Index 6

- $prev = a_5 = 0$.
- Where did 0 appear before index 5 ? Occurrences earlier: indices $0, 1, 3$. The **last prior** is $j = 3$.
- So $a_6 = (5) - (3) = 2$.
Calculation: $(n-1) = 5$, $j = 3 \Rightarrow 5 - 3 = 2$.

Index 7

- $\text{prev} = a_6 = 2$.
- Where did 2 appear before index 6? It appeared at index 4. So $j = 4$.
- So $a_7 = (6) - (4) = 2$.
Calculation: $(n-1) = 6, j = 4 \Rightarrow 6 - 4 = 2$.

Index 8

- $\text{prev} = a_7 = 2$.
- Where did 2 appear before index 7? It appeared at indices 4 and 6; the **last prior** is $j = 6$.
- So $a_8 = (7) - (6) = 1$.
Calculation: $(n-1) = 7, j = 6 \Rightarrow 7 - 6 = 1$.

Index 9

- $\text{prev} = a_8 = 1$.
- Where did 1 appear before index 8? It appeared at index 2. So $j = 2$.
- So $a_9 = (8) - (2) = 6$.
Calculation: $(n-1) = 8, j = 2 \Rightarrow 8 - 2 = 6$.

Summary table (compact)

n	a[n]	prev = a[n-1]	last prior index j (if any)	a[n] calculation
0	0	—	—	start = 0
1	0	0	none	0
2	1	0	0	$1 - 0 = 1$
3	0	1	none	0
4	2	0	1	$3 - 1 = 2$
5	0	2	none	0
6	2	0	3	$5 - 3 = 2$
7	2	2	4	$6 - 4 = 2$
8	1	2	6	$7 - 6 = 1$

n	a[n]	prev = a[n-1]	last prior index j (if any)	a[n] calculation
9	6	1	2	8 - 2 = 6

If you want, I can also show:

- the **exact map-based algorithm** (how to track the last seen indices in code step-by-step), or
- print the sequence while showing the `lastSeen` map state after each term so you can visualize how the map evolves. Which would you prefer?

give more OOP Programs (Builder, Factory Pattern) coding questions

Got it — I'll prepare **OOP-based Java coding questions** focusing on **Builder Pattern** and **Factory Pattern**, with problem statements, example inputs/outputs, and short explanations so they're interview-ready.

1. Builder Pattern — Create a `Computer` Object

Problem Statement:

Create a `Computer` class using the Builder Pattern so that you can flexibly construct a computer object with optional parameters like HDD, RAM, graphics card, and OS.

Example Input:

```
java

Computer comp = new Computer.Builder("500 GB", "8 GB")
    .setGraphicsCard(true)
    .setOS("Windows 11")
    .build();

System.out.println(comp);
```

Example Output:

```
nginx
```



```
Computer [HDD=500 GB, RAM=8 GB, GraphicsCard=true, OS=Windows 11]
```

Code:

java

```
class Computer {
    private String HDD;
    private String RAM;
    private boolean graphicsCard;
    private String OS;

    private Computer(Builder builder) {
        this.HDD = builder.HDD;
        this.RAM = builder.RAM;
        this.graphicsCard = builder.graphicsCard;
        this.OS = builder.OS;
    }

    public static class Builder {
        private String HDD;
        private String RAM;
        private boolean graphicsCard;
        private String OS;

        public Builder(String HDD, String RAM) {
            this.HDD = HDD;
            this.RAM = RAM;
        }

        public Builder setGraphicsCard(boolean graphicsCard) {
            this.graphicsCard = graphicsCard;
            return this;
        }

        public Builder setOS(String OS) {
            this.OS = OS;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}
```

```

@Override
public String toString() {
    return "Computer [HDD=" + HDD + ", RAM=" + RAM +
        ", GraphicsCard=" + graphicsCard + ", OS=" + OS
+ "]\n";
}

public class Main {
    public static void main(String[] args) {
        Computer comp = new Computer.Builder("500 GB", "8 GB")
            .setGraphicsCard(true)
            .setOS("Windows 11")
            .build();
        System.out.println(comp);
    }
}

```

2. Factory Pattern — Shape Factory

Problem Statement:

Create a `Shape` interface with multiple implementations (`Circle`, `Rectangle`, `Square`). Implement a `ShapeFactory` that returns an instance of the required shape.

Example Input:

```

java

ShapeFactory shapeFactory = new ShapeFactory();
Shape shape1 = shapeFactory.getShape("CIRCLE");
shape1.draw();

Shape shape2 = shapeFactory.getShape("RECTANGLE");
shape2.draw();

```

Example Output:

```

mathematica

Drawing Circle
Drawing Rectangle

```

Code:

```
java

interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Drawing Square");
    }
}

class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType == null) return null;
        if (shapeType.equalsIgnoreCase("CIRCLE")) return new
Circle();
        if (shapeType.equalsIgnoreCase("RECTANGLE")) return
new Rectangle();
        if (shapeType.equalsIgnoreCase("SQUARE")) return new
Square();
        return null;
    }
}

public class Main {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();
    }
}
```

```
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        shape2.draw();
    }
}
```

3. Factory Pattern with Parameterized Object Creation

Problem Statement:

Implement a `VehicleFactory` that returns either a `Car` or a `Bike` based on input. Both should have a `getSpeed()` method.

Example Input:

```
java

Vehicle car = VehicleFactory.getVehicle("car", 180);
Vehicle bike = VehicleFactory.getVehicle("bike", 120);

System.out.println(car.getSpeed());
System.out.println(bike.getSpeed());
```

Example Output:

```
bash

Car speed: 180 km/h
Bike speed: 120 km/h
```

Code:

```
java

interface Vehicle {
    String getSpeed();
}

class Car implements Vehicle {
    private int speed;
    public Car(int speed) { this.speed = speed; }
    public String getSpeed() { return "Car speed: " + speed +
    " km/h"; }
}

class Bike implements Vehicle {
```

```
private int speed;
public Bike(int speed) { this.speed = speed; }
public String getSpeed() { return "Bike speed: " + speed +
" km/h"; }
}

class VehicleFactory {
    public static Vehicle getVehicle(String type, int speed) {
        if (type.equalsIgnoreCase("car")) return new
Car(speed);
        if (type.equalsIgnoreCase("bike")) return new
Bike(speed);
        return null;
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = VehicleFactory.getVehicle("car", 180);
        Vehicle bike = VehicleFactory.getVehicle("bike", 120);

        System.out.println(car.getSpeed());
        System.out.println(bike.getSpeed());
    }
}
```

I can prepare **20+ OOP pattern coding problems** like this for your **Java Interview Master Kit** — including **Singleton, Prototype, Observer, Decorator, and Strategy Pattern** with examples and outputs.

Do you want me to **continue building the full OOP pattern section** now so it becomes part of your final 50+ program set?

File Handling (Read/Write, BufferedReader, Serialization)

Alright — let's make a **full set of Java coding questions and examples** for **File Handling** covering:

- **Basic File Read/Write**
- **BufferedReader / BufferedWriter**

- **Serialization / Deserialization**

I'll prepare them with **question + explanation + code** so they can be used for interviews or practice.

1. File Read & Write using `FileWriter` and `FileReader`

Question:

Write a Java program to create a file named `sample.txt`, write `"Hello World"` to it, and read it back.

Code:

```
java

import java.io.*;

public class FileReadWriteExample {
    public static void main(String[] args) {
        String filename = "sample.txt";

        // Writing to file
        try (FileWriter fw = new FileWriter(filename)) {
            fw.write("Hello World");
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reading from file
        try (FileReader fr = new FileReader(filename)) {
            int ch;
            while ((ch = fr.read()) != -1) {
                System.out.print((char) ch);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Read a File using `BufferedReader`

Question:

Read a text file `data.txt` and print each line to the console.

Code:

```
java

import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        String filename = "data.txt";

        try (BufferedReader br = new BufferedReader(new
        FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- `BufferedReader` reads text efficiently by buffering the characters.
- `readLine()` returns a `String` until it reaches the end of file.

3. Write to File using `BufferedWriter`

Question:

Write multiple lines to a file `output.txt` using `BufferedWriter`.

Code:

```
java

import java.io.*;

public class BufferedWriterExample {
    public static void main(String[] args) {
        String filename = "output.txt";
```

```

        try (BufferedWriter bw = new BufferedWriter(new
FileWriter(filename))) {
            bw.write("Line 1: Hello");
            bw.newLine();
            bw.write("Line 2: World");
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

4. Object Serialization

Question:

Serialize a `Person` object to a file `person.ser`.

Code:

```

java

import java.io.*;

class Person implements Serializable {
    private String name;
    private int age;

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

    public String toString() {
        return "Name: " + name + ", Age: " + age;
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        Person p = new Person("John", 25);

        try (ObjectOutputStream oos = new
ObjectOutputStream(new FileOutputStream("person.ser"))) {
            oos.writeObject(p);

```



```
        System.out.println("Person object serialized.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

5. Object Deserialization

Question:

Read back the serialized object from `person.ser` and display it.

Code:

```
java

import java.io.*;

public class DeserializationExample {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("person.ser"))) {
            Person p = (Person) ois.readObject();
            System.out.println("Deserialized: " + p);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

6. Count Words in a File

Question:

Write a program to count the number of words in `data.txt`.

Code:

```
java

import java.io.*;

public class WordCountExample {
```

```
public static void main(String[] args) {  
    String filename = "data.txt";  
    int wordCount = 0;  
  
    try (BufferedReader br = new BufferedReader(new  
        FileReader(filename))) {  
        String line;  
        while ((line = br.readLine()) != null) {  
            wordCount += line.split("\\s+").length;  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
  
    System.out.println("Total Words: " + wordCount);  
}
```

7. Copy File Content

Question:

Write a Java program to copy data from `source.txt` to `destination.txt`.

Code:

```
java  
  
import java.io.*;  
  
public class FileCopyExample {  
    public static void main(String[] args) {  
        try (FileInputStream fis = new  
            FileInputStream("source.txt");  
            FileOutputStream fos = new  
            FileOutputStream("destination.txt")) {  
  
            byte[] buffer = new byte[1024];  
            int bytesRead;  
  
            while ((bytesRead = fis.read(buffer)) != -1) {  
                fos.write(buffer, 0, bytesRead);  
            }  
        }  
    }  
}
```

```
        System.out.println("File copied successfully.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

If you want, I can **add 10 more advanced file handling programs** like:

- Append to a file
- Merge two files
- Search for a word in a file
- Count lines in a file
- Split large file into smaller files
- Read CSV file
- Write CSV file
- JSON read/write
- Directory listing and file filtering
- Monitor file changes in real time

Do you want me to prepare **this extended advanced set** so your file handling kit will be 15+ programs?

explain serialization and deserailization with respect to spring boot rest api's

Alright — let's break down **serialization** and **deserialization** first, and then I'll connect them to **Spring Boot REST APIs** with an example.

1. What is Serialization and Deserialization?

Concept	Meaning	Example in Java
Serialization	Converting a Java object into a byte stream (or JSON/XML in REST context) so it can be stored or transferred.	<code>User</code> object → JSON string → sent in HTTP response.

Concept	Meaning	Example in Java
Deserialization	Converting the byte stream (or JSON/XML) back into a Java object.	JSON string from request body → <code>User</code> object.

In a **Spring Boot REST API**:

- **Serialization** happens when your API **returns** an object, and Spring Boot (via Jackson by default) converts it to JSON for the client.
- **Deserialization** happens when your API **receives** JSON in a request body and converts it into a Java object.

2. How It Works in Spring Boot

Spring Boot uses **Jackson** (a JSON processing library) by default for:

- `@RequestBody` → **Deserialization**
- `@ResponseBody` → **Serialization**

3. Example

`User.java`

```
java

package com.example.demo.model;

import java.io.Serializable;

public class User implements Serializable {
    private String name;
    private int age;

    // Constructors
    public User() {}
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getters & Setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
```

```
public int getAge() { return age; }  
public void setAge(int age) { this.age = age; }  
}
```

UserController.java

```
java  
  
package com.example.demo.controller;  
  
import com.example.demo.model.User;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
@RequestMapping("/users")  
public class UserController {  
  
    // POST → JSON to Java Object (Deserialization)  
    @PostMapping  
    public String createUser(@RequestBody User user) {  
        return "User created: " + user.getName() + ", Age: " +  
user.getAge();  
    }  
  
    // GET → Java Object to JSON (Serialization)  
    @GetMapping("/{name}")  
    public User getUser(@PathVariable String name) {  
        return new User(name, 25); // Returning Java object,  
will be converted to JSON  
    }  
}
```

4. Step-by-Step Flow

A. Deserialization (Client → Server)

1. Client sends:

```
json  
  
{  
    "name": "Alice",
```

```
"age": 30  
}
```

2. Spring Boot uses **Jackson** to convert JSON → `User` object.
 3. `createUser()` method receives a **fully populated** `User` instance.
-

B. Serialization (Server → Client)

1. You return a `User` object from `getUser()`.
2. Spring Boot uses **Jackson** to convert `User` → JSON:

```
json  
  
{  
  "name": "Bob",  
  "age": 25  
}
```

3. JSON is sent in the HTTP response.
-

5. Serialization in Binary Form

If you need to store the object as a **byte stream** (e.g., in a file or cache):

```
java  
  
import java.io.*;  
  
public class SerializeExample {  
    public static void main(String[] args) throws IOException,  
        ClassNotFoundException {  
        User user = new User("Charlie", 40);  
  
        // Serialize to file  
        ObjectOutputStream oos = new ObjectOutputStream(new  
        FileOutputStream("user.ser"));  
        oos.writeObject(user);  
        oos.close();  
  
        // Deserialize from file  
        ObjectInputStream ois = new ObjectInputStream(new  
        FileInputStream("user.ser"));  
        User deserializedUser = (User) ois.readObject();  
        ois.close();  
    }  
}
```

```
        System.out.println("Deserialized: " +  
        deserializedUser.getName() + ", Age: " +  
        deserializedUser.getAge());  
    }  
}
```

✅ In REST APIs, we rarely use binary serialization directly — we use **JSON serialization/deserialization via Jackson**, which happens automatically with `@RequestBody` and `@ResponseBody`.

If you want, I can give you a **Spring Boot REST API example where binary serialization/deserialization is actually used**, for example storing API responses in `.ser` files for caching. That would show **both JSON and binary serialization** in one flow.

where `@ResponseBody` used?

In **Spring Boot**, `@ResponseBody` is used to tell **Spring MVC** that the return value of a method should be written directly to the HTTP response body **instead of** rendering a view (like a JSP, Thymeleaf page, etc.).

Where it's used

- On **controller methods** to directly send JSON, XML, plain text, or any other object as the HTTP response.
 - Often used with **REST APIs**, since they usually return JSON or XML instead of HTML templates.
 - Can be placed **on a method** or on a **controller class** (to apply to all methods in that controller).
-

How it works

When you use `@ResponseBody`:

1. Spring **bypasses** the view resolution mechanism.

2. The returned object is **automatically serialized** (using **Jackson** by default) into JSON or XML and written into the response body.
 3. The `Content-Type` of the response is set based on the `produces` attribute or the request's `Accept` header.
-

Example 1 – Method Level

```
java

@RestController
@RequestMapping("/api")
public class UserController {

    @GetMapping("/user")
    @ResponseBody
    public User getUser() {
        return new User("John", "Doe");
    }
}
```

Here:

- The method returns a `User` object.
 - `@ResponseBody` tells Spring to **convert** that object into JSON (via Jackson) and **send it as HTTP response**.
-

Example 2 – Class Level

Instead of writing `@ResponseBody` for every method, you can use `@RestController` :

```
java

@RestController // @Controller + @ResponseBody
@RequestMapping("/api")
public class UserController {

    @GetMapping("/user")
    public User getUser() {
        return new User("John", "Doe");
    }
}
```


- `@RestController` is equivalent to using `@Controller` + `@ResponseBody` on every method in the class.

When you DON'T need it

- If your class is already annotated with `@RestController`.
- If you're returning a view (e.g., JSP or Thymeleaf), in which case you should **not** use `@ResponseBody`.

✅ In short:

- `@ResponseBody` → Directly writes method return value to HTTP response body (JSON, XML, String, etc.).
- `@RestController` → Automatically applies `@ResponseBody` to all methods in that controller.

If you want, I can also show **how serialization & deserialization works together with** `@ResponseBody` inside a Spring Boot REST API. That will connect the dots with your earlier question.

Patterns & Advanced (Observer, Strategy, Producer-Consumer)

Got it — you want **Patterns & Advanced** coding questions, specifically around **Observer**, **Strategy**, and **Producer-Consumer**.

Here's a breakdown with **coding examples** and **explanations** so you can see exactly how they work in practice.

1. Observer Pattern

Use Case: One-to-many dependency — when one object changes state, all its dependents are notified automatically.

Example: Weather updates being sent to multiple display boards.

Code

```
java
```

```
// Subject interface
interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

// Observer interface
interface Observer {
    void update(String message);
}

// Concrete Subject
class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(news);
        }
    }
}

// Concrete Observer
class NewsChannel implements Observer {
    private String name;
    public NewsChannel(String name) { this.name = name; }
```

```

@Override
public void update(String message) {
    System.out.println(name + " received news: " +
message);
}
}

// Test
public class ObserverExample {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();
        Observer cnn = new NewsChannel("CNN");
        Observer bbc = new NewsChannel("BBC");

        agency.registerObserver(cnn);
        agency.registerObserver(bbc);

        agency.setNews("Elections are coming!");
        agency.setNews("Weather is sunny today!");
    }
}

```

2. Strategy Pattern

Use Case: Define a family of algorithms, encapsulate each one, and make them interchangeable.

Example: Different payment methods in an e-commerce app.

Code

```

java

// Strategy interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit
Card.");
    }
}

```

```

    }

    class PayPalPayment implements PaymentStrategy {
        @Override
        public void pay(int amount) {
            System.out.println("Paid " + amount + " using
PayPal.");
        }
    }

    // Context
    class PaymentContext {
        private PaymentStrategy strategy;

        public PaymentContext(PaymentStrategy strategy) {
            this.strategy = strategy;
        }

        public void executePayment(int amount) {
            strategy.pay(amount);
        }
    }

    // Test
    public class StrategyExample {
        public static void main(String[] args) {
            PaymentContext context = new PaymentContext(new
CreditCardPayment());
            context.executePayment(1000);

            context = new PaymentContext(new PayPalPayment());
            context.executePayment(2000);
        }
    }

```

3. Producer-Consumer Pattern

Use Case: Synchronization between threads — one produces data, another consumes it.

Example: Job queue processing.

Code

```
java
```

```
import java.util.LinkedList;
import java.util.Queue;

class SharedBuffer {
    private Queue<Integer> queue = new LinkedList<>();
    private int capacity = 5;

    public synchronized void produce(int value) throws
InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }
        queue.add(value);
        System.out.println("Produced: " + value);
        notifyAll();
    }

    public synchronized int consume() throws
InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        int value = queue.poll();
        System.out.println("Consumed: " + value);
        notifyAll();
        return value;
    }
}

class Producer extends Thread {
    private SharedBuffer buffer;

    public Producer(SharedBuffer buffer) { this.buffer =
buffer; }

    public void run() {
        int value = 0;
        try {
            while (true) {
                buffer.produce(value++);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }

    class Consumer extends Thread {
        private SharedBuffer buffer;

        public Consumer(SharedBuffer buffer) { this.buffer =
buffer; }

        public void run() {
            try {
                while (true) {
                    buffer.consume();
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
e.printStackTrace(); }
        }
    }

    // Test
    public class ProducerConsumerExample {
        public static void main(String[] args) {
            SharedBuffer buffer = new SharedBuffer();
            Producer producer = new Producer(buffer);
            Consumer consumer = new Consumer(buffer);

            producer.start();
            consumer.start();
        }
    }

```

✅ If you want, I can now prepare a list of 30+ advanced OOP pattern coding questions mixing Builder, Factory, Observer, Strategy, Producer-Consumer, Singleton, Prototype, Adapter, Decorator, etc., so you get a full interview set.

Do you want me to go ahead with that list?

give more coding examples

Got it — you want more advanced Java coding examples for patterns like Observer, Strategy, and Producer-Consumer.

I'll give you three **complete, runnable examples** for each pattern, with **step-by-step explanations** so you can easily adapt them for interviews or projects.

1. Observer Pattern – Example: News Agency & Subscribers

Concept: When one object changes state, all its dependents are notified automatically.

```
java

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

// Observer interface
interface Observer {
    void update(String news);
}

// Concrete Observer
class NewsSubscriber implements Observer {
    private String name;
    public NewsSubscriber(String name) { this.name = name; }
    @Override
    public void update(String news) {
        System.out.println(name + " received news update: " +
news);
    }
}

// Subject interface
interface Subject {
    void addObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

// Concrete Subject
class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String latestNews;

    @Override
```

```

    public void addObserver(Observer o) { observers.add(o); }
    @Override
    public void removeObserver(Observer o) {
observers.remove(o); }
    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(latestNews);
        }
    }

    public void setLatestNews(String news) {
        this.latestNews = news;
        notifyObservers();
    }
}

public class ObserverPatternExample {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();

        Observer o1 = new NewsSubscriber("Alice");
        Observer o2 = new NewsSubscriber("Bob");

        agency.addObserver(o1);
        agency.addObserver(o2);

        agency.setLatestNews("Breaking News: Java 21
Released!");
        agency.setLatestNews("Weather Update: Sunny
Tomorrow");
    }
}

```

How it works:

- **Publisher (NewsAgency)** sends updates to all **subscribers (NewsSubscriber)** when data changes.
- New subscribers can be added/removed at runtime.

2. Strategy Pattern – Example: Payment Methods

Concept: Define a family of algorithms, encapsulate each, and make them interchangeable at runtime.


```
java

// Strategy Interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}

// Context
class ShoppingCart {
    private PaymentStrategy paymentStrategy;
    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }
    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}

public class StrategyPatternExample {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategy(new CreditCardPayment());
        cart.checkout(500);

        cart.setPaymentStrategy(new PayPalPayment());
        cart.checkout(1200);
    }
}
```

```
    }
}
```

How it works:

- You can switch payment methods at runtime without changing the main cart logic.

3. Producer-Consumer Pattern – Example: Message Queue

Concept: One or more producers generate data, and one or more consumers process it. Uses a **BlockingQueue** to avoid race conditions.

```
java

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

// Producer
class Producer implements Runnable {
    private BlockingQueue<String> queue;
    public Producer(BlockingQueue<String> queue) { this.queue
= queue; }

    @Override
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                String message = "Message " + i;
                System.out.println("Produced: " + message);
                queue.put(message);
                Thread.sleep(500);
            }
            queue.put("DONE");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); }
    }
}

// Consumer
class Consumer implements Runnable {
    private BlockingQueue<String> queue;
    public Consumer(BlockingQueue<String> queue) { this.queue
= queue; }
```

```

@Override
public void run() {
    try {
        String msg;
        while (!(msg = queue.take()).equals("DONE")) {
            System.out.println("Consumed: " + msg);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

public class ProducerConsumerExample {
    public static void main(String[] args) {
        BlockingQueue<String> queue = new
        LinkedBlockingQueue<>();

        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        producer.start();
        consumer.start();
    }
}

```

How it works:

- **BlockingQueue** automatically handles thread synchronization.
- Producer creates messages and puts them in the queue.
- Consumer takes messages and processes them.

I can also prepare **Builder Pattern + Factory Pattern with full examples** so you have a **complete OOP patterns cheat sheet** for interviews.
Do you want me to prepare those next?