Core java Collection framework and oops:

Of course! Here's a **Core Java Interview Questions and Answers** guide — covering topics like String, Builder, Buffer, immutability, == vs equals(), JVM, JDK, and OOP concepts.

What is serializable?  
a,

**Serializable:**

* **Definition**: The Serializable interface in Java is a marker interface (it doesn't contain any methods) that indicates a class can be serialized. Serialization is the process of converting an object's state (its instance variables) into a byte stream so that it can be saved to a file, sent over a network, or stored in memory.

package javafiles;  
import java.io.\*;  
  
public class SerialDemo {  
 public static void main(String[] args) throws Exception {  
 Save obj = new Save();  
 obj.i = 4;  
 // Serialization: Writing object to file  
 File f = new File("Obj.txt");  
 FileOutputStream fos = new FileOutputStream(f);  
 ObjectOutputStream oos = new ObjectOutputStream(fos);  
 oos.writeObject(obj);  
 oos.close();  
 // Deserialization: Reading object from file  
 FileInputStream fis = new FileInputStream(f);  
 ObjectInputStream ois = new ObjectInputStream(fis);  
 Save obj1 = (Save) ois.readObject();  
 ois.close();  
 // Printing the deserialized object's value  
 System.*out*.println("Value of Obj1: " + obj1.i);  
 }  
};  
class Save implements Serializable { // Implement Serializable interface  
 int i;  
}

5. What is the transient keyword in Java?

• Used to indicate that a variable should not be serialized.

**transient:**

* **Definition**: The transient keyword in Java is used to indicate that a field in a class should not be serialized. When an object is serialized, any fields marked as transient are ignored and not written to the output stream.

Example:

package javafiles;  
import java.io.\*;  
  
public class Serialtransient {  
 public static void main(String[] args) throws Exception {  
 Save1 obj = new Save1();  
 obj.i = 4;  
 obj.j = 10; // Adding a new field `j`  
  
 // Serialization: Writing object to file  
 File f = new File("Obj1.txt");  
 FileOutputStream fos = new FileOutputStream(f);  
 ObjectOutputStream oos = new ObjectOutputStream(fos);  
 oos.writeObject(obj);  
 oos.close();  
  
 // Deserialization: Reading object from file  
 FileInputStream fis = new FileInputStream(f);  
 ObjectInputStream ois = new ObjectInputStream(fis);  
 Save1 obj1 = (Save1) ois.readObject();  
 ois.close();  
  
 // Printing the deserialized object's values  
 System.*out*.println("Value of obj1.i: " + obj1.i); // This will be printed  
 System.*out*.println("Value of obj1.j: " + obj1.j); // This will be 0 because `j` is transient  
 }  
};  
  
class Save1 implements Serializable {  
 int i;  
 transient int j; // Marking field `j` as transient  
}

* **🔹 1. String, StringBuilder, StringBuffer**

**Q1. What is the difference between String, StringBuilder, and StringBuffer?**

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| Mutability | Immutable | Mutable | Mutable |
| Thread-safe | No | No | Yes (synchronized) |
| Performance | Slow in concatenation | Fast | Slower than StringBuilder |

**Q2. Why is String immutable in Java?**

* Because once a String object is created, it cannot be changed.
* **Reasons:**
  + Security (used in class loading, file paths, etc.)
  + Thread safety (safe to use across threads)
  + Caching in string pool (string literals are shared)

**Q3. How does StringBuilder differ from StringBuffer?**

* Both are mutable, but:
  + StringBuffer is **synchronized** → thread-safe but slower.
  + StringBuilder is **not synchronized** → faster but not thread-safe.

**Q4. What happens in:**

String s1 = "abc";

String s2 = s1 + "def";

* s2 creates a **new String object** ("abcdef") since String is immutable. s1 remains "abc".

**🔹 2. == vs equals()**

**Q1. What is the difference between == and .equals()?**

* == checks **reference** (memory address).
* .equals() checks **content/value**.

**Q2. Example:**

String s1 = "hello";

String s2 = new String("hello");

System.out.println(s1 == s2); // false (different memory)

System.out.println(s1.equals(s2)); // true (same content)

**Q3. What is String interning?**

* JVM maintains a **string pool**.
* If a string with the same content exists, new variables point to the same object to save memory.

**🔹 3. JVM, JDK, JRE**

**Q1. What is the difference?**

| **Term** | **Description** |
| --- | --- |
| **JVM (Java Virtual Machine)** | Runs the Java bytecode (.class files) |
| **JRE (Java Runtime Environment)** | JVM + libraries to run Java apps |
| **JDK (Java Development Kit)** | JRE + tools for development (compiler, debugger, etc.) |

**Q2. JVM Memory Areas:**

* **Heap**: Objects
* **Stack**: Method calls and local variables
* **Method Area**: Class metadata
* **PC Register**: Address of current instruction
* **Native Method Stack**: Native (C/C++) method calls

**Q3. What is JIT (Just-In-Time) Compiler?**

* A part of JVM that compiles bytecode to native code at runtime to improve performance.

**Q4,🔹== vs .equals() in Java**

| **Feature** | **== Operator** | **.equals() Method** |
| --- | --- | --- |
| **Type** | **Operator** | **Method (defined in Object class)** |
| **Checks** | **Reference equality (memory address)** | **Content equality (data/values)** |
| **Applicable to** | **Primitives & Objects** | **Objects only** |
| **Customizable** | **❌ No** | **✅ Yes (can override in your own classes)** |

Q5,

Absolutely! Here's a **deep dive into Exception Handling in Java** — super important for interviews and real-world development. We'll cover concepts, types, keywords, best practices, and code examples.

**🚨 What is Exception Handling in Java?**

**Exception handling** is a mechanism to handle **runtime errors** so that the normal flow of the application can be maintained.

**🔹 1. Exception Hierarchy**

Throwable

├── Exception (checked)

│ ├── IOException

│ ├── SQLException

│ └── etc.

└── RuntimeException (unchecked)

├── NullPointerException

├── ArithmeticException

└── etc.

* Throwable is the superclass for all errors and exceptions.
* Exception: conditions that applications might want to catch.
* Error: serious problems (like OutOfMemoryError) — not meant to be caught.

**🔹 2. Types of Exceptions**

| **Type** | **Description** | **Examples** |
| --- | --- | --- |
| **Checked Exception** | Caught or declared in the method signature | IOException, SQLException |
| **Unchecked Exception** | Occur during runtime | NullPointerException, ArithmeticException |
| **Error** | Serious issues beyond app control | OutOfMemoryError, StackOverflowError |

**🔹 3. Exception Handling Keywords**

| **Keyword** | **Use** |
| --- | --- |
| try | Wrap code that might throw an exception |
| catch | Handle the exception |
| finally | Executes code regardless of exception |
| throw | Throw an exception manually |
| throws | Declare exceptions in method signature |

**🔸 Example:**

try {

int a = 10 / 0; // ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero: " + e);

} finally {

System.out.println("This block always executes.");

}

**🔹 4. throw vs throws**

| **throw** | **throws** |
| --- | --- |
| Used to explicitly throw an exception | Used in method signature |
| Only one exception can be thrown | Can declare multiple exceptions |
| Followed by instance | Followed by class name |

**Example:**

void checkAge(int age) {

if (age < 18)

throw new IllegalArgumentException("Underage not allowed");

}

void readFile() throws IOException {

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

}

**🔹 5. Multiple Catch Blocks**

try {

String s = null;

System.out.println(s.length());

} catch (NullPointerException e) {

System.out.println("Null Pointer caught!");

} catch (Exception e) {

System.out.println("General exception caught.");

}

**Note**: Always catch subclasses first. Catching Exception before NullPointerException will cause a compile error.

**🔹 6. finally Block Use Case**

Used for:

* Closing DB connections
* Closing file readers
* Cleanup code

try {

int[] arr = new int[3];

arr[5] = 10;

} catch (ArrayIndexOutOfBoundsException e) {

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

} finally {

System.out.println("Cleanup done");

}

**🔹 7. Custom Exception**

You can define your own exceptions by extending Exception or RuntimeException.

class MyException extends Exception {

MyException(String msg) {

super(msg);

}

}

public void validate(int age) throws MyException {

if (age < 18)

throw new MyException("Not eligible");

}

**🔹 8. Best Practices**

* Don’t catch generic Exception unless needed.
* Close resources using **try-with-resources** (Java 7+).
* Use custom exceptions for business logic.
* Avoid swallowing exceptions silently.
* Log exceptions properly for debugging.

**🔹 9. try-with-resources (Java 7+)**

Auto-close resources like BufferedReader, FileReader.

try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {

String line = br.readLine();

} catch (IOException e) {

e.printStackTrace();

}

**🔹 10. Exception Propagation**

If not caught in the current method, the exception **bubbles up** to the caller.

public void a() {

b(); // exception in b() propagates to a()

}

public void b() {

int x = 10 / 0;

}

Some core java interview questions from backend developer?  
A,

**Q1. Why is Java not 100% object-oriented?** Java is not considered 100% object-oriented because it uses primitive data types (like int, char, boolean, etc.) which are not objects. This is done for performance reasons.

**Q2. Why are pointers not used in Java?** Pointers are not used in Java to ensure security and simplicity. Java uses references instead, which do not allow direct memory access, thus preventing potential security issues and making the language easier to use.

**Q3. What is the JIT compiler in Java?** The Just-In-Time (JIT) compiler is a part of the Java Runtime Environment (JRE) that improves the performance of Java applications by compiling bytecode into native machine code at runtime.

**Q4. What is the "Write Once, Run Anywhere" concept in Java?** This concept means that Java programs can be written on one platform and run on any other platform without modification, thanks to the Java Virtual Machine (JVM) which abstracts the underlying hardware and operating system.

**Q5. What is a marker interface in Java?** A marker interface is an interface with no methods or fields. It is used to signal to the JVM or other code that a class implementing the interface has a certain property. Examples include Serializable and Cloneable.

**Q6. Does finally always execute in Java?** The finally block always executes when the try block exits, except when the JVM exits due to a System.exit() call or if the thread executing the try block is interrupted or killed.

**Q7. How to make a class immutable in Java?** To make a class immutable:

1. Declare the class as final so it cannot be subclassed.
2. Make all fields private and final.
3. Do not provide setters.
4. Initialize all fields via a constructor.
5. Ensure that mutable fields are not directly accessible and return copies of mutable objects.

**Q8. Does Java support multiple inheritance?** Java does not support multiple inheritance with classes to avoid complexity and simplify the design. However, it supports multiple inheritance through interfaces.

**Q9. Can we serialize static variables in Java?** No, static variables belong to the class, not instances, and are not serialized. Serialization is concerned with the state of an object.

**Q10. Can we use the default constructor if an explicit constructor is defined?** No, if an explicit constructor is defined, the default constructor is not provided by the compiler. You must define it explicitly if needed.

**Q11. Can we rethrow an exception in Java?** Yes, you can rethrow an exception using the throw keyword within a catch block.

**Q12. What is the purpose of the transient keyword in Java?** The transient keyword is used to indicate that a field should not be serialized. When an object is serialized, transient fields are not included in the serialized representation.

**Q13. Why are Comparable and Comparator interfaces required?**

* Comparable is used to define the natural ordering of objects.
* Comparator is used to define custom orderings of objects.

**Q14. Comparable implementation in Java?**

public class Student implements Comparable<Student> {

private String name;

private int age;

public int compareTo(Student other) {

return this.age - other.age; // Natural ordering by age

}

}

**Q15. Comparator implementation in Java?**

import java.util.Comparator;

public class NameComparator implements Comparator<Student> {

public int compare(Student s1, Student s2) {

return s1.getName().compareTo(s2.getName()); // Custom ordering by name

}

}

**Q16. What is the difference between Comparator and Comparable?**

* Comparable is used for natural ordering and is implemented by the class itself.
* Comparator is used for custom ordering and is implemented by a separate class.

**Q17. Can we override static methods in Java?** No, static methods cannot be overridden because they belong to the class, not instances. They can be hidden by defining a static method with the same signature in a subclass.

**Q18. What is the final keyword in Java?** The final keyword can be used with classes, methods, and variables:

* **Class**: Prevents the class from being subclassed.
* **Method**: Prevents the method from being overridden.
* **Variable**: Makes the variable constant (its value cannot be changed once assigned).

**Q19. What is the finalize() method?** The finalize() method is called by the garbage collector before an object is destroyed. It is used to perform cleanup operations. However, its use is generally discouraged in favor of other resource management techniques.

**Q20. Difference between throw and throws?**

* throw: Used to explicitly throw an exception.
* throws: Used in a method signature to declare that the method can throw specified exceptions.

Feel free to ask if you have more questions or need further clarification on any of these topics!

**Q21. What is a serialVersionUID and why is it important?** serialVersionUID is a unique identifier for each Serializable class. It is used during the deserialization process to ensure that a loaded class corresponds exactly to a serialized object. If no matching serialVersionUID is found, an InvalidClassException is thrown.

**Q22. Why is 1 == 1 true but 1000 == 1000 false when dealing with Wrapper Classes in Java?** This behavior is due to Java's integer caching. For values between -128 and 127, Java caches Integer objects, so == returns true for these values. For values outside this range, new Integer objects are created, so == returns false as it compares object references, not values.

**Q23. What are variable arguments in Java?** Variable arguments (varargs) allow a method to accept zero or more arguments of a specified type. They are represented by three dots (...) in the method signature.

public void printNumbers(int... numbers) {

for (int number : numbers) {

System.out.println(number);

}

}

**Q24. Where should we use Enum?** Enums are used to define a fixed set of constants. They are useful for representing predefined values like days of the week, directions, states, etc.

public enum Day {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

**Q25. How do you initialize a BigInteger?** You can initialize a BigInteger using a string representation of the number or an integer value.

BigInteger bigInt1 = new BigInteger("12345678901234567890");

BigInteger bigInt2 = BigInteger.valueOf(1234567890L);

**Q26. Why can't final and abstract be used together in Java?** final and abstract have opposite meanings. final means the class cannot be subclassed, while abstract means the class must be subclassed to provide implementations for abstract methods. Therefore, they cannot be used together.

**Q27. Does an abstract class need to have an abstract method?** No, an abstract class does not need to have an abstract method. However, it cannot be instantiated and is typically used to provide a common base for subclasses.

**Q28. Can constructors be synchronized in Java?** No, constructors cannot be synchronized because it would lock the object being created, which is not fully formed yet. However, you can synchronize the code inside the constructor.

**Q29. If Class B extends Class A, what will be the order of execution of static initializers, instance initializers, and constructors?** The order of execution is:

1. Static initializers of Class A.
2. Static initializers of Class B.
3. Instance initializers and constructors of Class A.
4. Instance initializers and constructors of Class B.

**Q30. In what situations is the super keyword employed?** The super keyword is used to:

* Call the superclass constructor.
* Access superclass methods that have been overridden.
* Access superclass fields that have been hidden.

**Q31. How to write multiple catch statements under a single try block?** You can write multiple catch statements to handle different exceptions.

try {

// Code that may throw exceptions

} catch (IOException e) {

// Handle IOException

} catch (SQLException e) {

// Handle SQLException

}

**Q32. What is implicit casting or widening conversion?** Implicit casting (widening conversion) is the automatic conversion of a smaller data type to a larger data type.

int i = 100;

long l = i; // Implicit casting from int to long

**Q33. What is explicit casting or narrowing conversion?** Explicit casting (narrowing conversion) is the manual conversion of a larger data type to a smaller data type.

long l = 100L;

int i = (int) l; // Explicit casting from long to int

**Q34. Which data type would you choose for storing currency values like Trading Price: Float, Double, or BigDecimal?** BigDecimal is preferred for storing currency values because it provides precise control over decimal numbers and avoids rounding errors associated with float and double.

**Q35. How can Java Reflection be used to break encapsulation?** Java Reflection can be used to access private fields and methods of a class, thus breaking encapsulation.

Field privateField = SomeClass.class.getDeclaredField("privateField");

privateField.setAccessible(true);

privateField.set(someObject, newValue);

**Q36. How do you decide if an inner class is required?** An inner class is required when:

* It logically belongs to the outer class.
* It needs access to the outer class's members.
* It is not intended to be used outside the outer class.

**Q37. How does the static keyword impact garbage collection in Java?** Static fields and methods belong to the class, not instances. They are not eligible for garbage collection until the class is unloaded, which typically happens when the JVM shuts down.

**Q38. How to choose initial capacity in an ArrayList constructor when the list is repeatedly cleared and reused?** Choose an initial capacity based on the maximum expected size of the list to minimize resizing operations. If the list size varies significantly, consider using a default capacity and allowing the list to resize dynamically.

**Q39. What are the potential issues with using mutable objects as keys in a HashMap?** Using mutable objects as keys can lead to inconsistent behavior if the object's state changes while it is being used as a key. This can cause issues with hash code calculations and equality checks, leading to incorrect retrievals.

**Q40. How do you show a diamond problem with multiple inheritance in Java?** Java avoids the diamond problem by not allowing multiple inheritance with classes. However, it can occur with interfaces. The diamond problem arises when a class inherits from two interfaces that have a common ancestor.

interface A {

void display();

}

interface B extends A {

void display();

}

interface C extends A {

void display();

}

class D implements B, C {

public void display() {

// Must provide implementation to resolve ambiguity

}

}

**Q41. How do you illustrate a try-catch example in Java?**

try {

int result = 10 / 0; // This will throw ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero: " + e.getMessage());

}

**Q42. Write a Java program to show a NullPointerException.**

public class NullPointerExample {

public static void main(String[] args) {

String str = null;

System.out.println(str.length()); // This will throw NullPointerException

}

}

**Q43. Show an example of switch expressions and multi-label case statements in Java.**

public class SwitchExample {

public static void main(String[] args) {

int day = 3;

switch (day) {

case 1, 2, 3 -> System.out.println("Weekday");

case 4, 5 -> System.out.println("Weekend");

default -> System.out.println("Invalid day");

}

}

}

**Q44. How do you write an interface with default and static methods?**

public interface MyInterface {

void abstractMethod();

default void defaultMethod() {

System.out.println("Default method");

}

static void staticMethod() {

System.out.println("Static method");

}

}

**Q45. How do you create a functional interface?** A functional interface is an interface with a single abstract method.

@FunctionalInterface

public interface MyFunctionalInterface {

void singleAbstractMethod();

}

**Q46. Show an example of using lambda expressions in Java.**

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println(name));

**Q47. Show examples of overloading and overriding in Java.**

* **Overloading**:

class Example {

void display(int a) {

System.out.println("Display with int: " + a);

}

void display(String a) {

System.out.println("Display with String: " + a);

}

}

* **Overriding**:

class Parent {

void display() {

System.out.println("Parent display");

}

}

class Child extends Parent {

@Override

void display() {

System.out.println("Child display");

}

}

**Q48. Sort the array in descending order.**

int[] arr = {5, 2, 8, 1, 3};

Arrays.sort(arr);

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;

}

**Q49. Find the sum of digits in a given string.**

String str = "a1b2c3";

int sum = 0;

for (char ch : str.toCharArray()) {

if (Character.isDigit(ch)) {

sum += Character.getNumericValue(ch);

}

}

System.out.println("Sum of digits

**Q50. Find the sum of digits in a given string.**

String str = "a1b2c3";

int sum = 0;

for (char ch : str.toCharArray()) {

if (Character.isDigit(ch)) {

sum += Character.getNumericValue(ch);

}

}

System.out.println("Sum of digits: " + sum);

**Q51. Reverse a string without using a third variable.**

String str = "hello";

char[] chars = str.toCharArray();

int left = 0, right = chars.length - 1;

while (left < right) {

chars[left] ^= chars[right];

chars[right] ^= chars[left];

chars[left] ^= chars[right];

left++;

right--;

}

String reversed = new String(chars);

System.out.println("Reversed string: " + reversed);

**Q52. Find the 5th largest number present in an array.**

int[] arr = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

Arrays.sort(arr);

int fifthLargest = arr[arr.length - 5];

System.out.println("5th largest number: " + fifthLargest);

**Q53. Find out whether two strings are identical or not.**

String str1 = "hello";

String str2 = "hello";

boolean areIdentical = str1.equals(str2);

System.out.println("Strings are identical: " + areIdentical);

**Q54. Count the number of matching characters in the pair of strings.**

String str1 = "hello";

String str2 = "world";

int count = 0;

for (char ch : str1.toCharArray()) {

if (str2.indexOf(ch) != -1) {

count++;

}

}

System.out.println("Number of matching characters: " + count);

**Q55. Generate the largest number possible using these digits.**

int[] digits = {3, 1, 4, 1, 5, 9};

Arrays.sort(digits);

StringBuilder largestNumber = new StringBuilder();

for (int i = digits.length - 1; i >= 0; i--) {

largestNumber.append(digits[i]);

}

System.out.println("Largest number: " + largestNumber);

**Q56. Check if given strings are rotations of each other or not.**

String str1 = "ABCD";

String str2 = "CDAB";

boolean isRotation = (str1.length() == str2.length()) && ((str1 + str2).contains(str2));

System.out.println("Strings are rotations: " + isRotation);

**Q57. How to add two large numbers in Java?**

BigInteger num1 = new BigInteger("12345678901234567890");

BigInteger num2 = new BigInteger("98765432109876543210");

BigInteger sum = num1.add(num2);

System.out.println("Sum: " + sum);

**Q58. How to swap two numbers without using a third variable?**

int a = 5, b = 10;

a = a + b;

b = a - b;

a = a - b;

System.out.println("Swapped values: a = " + a + ", b = " + b);

**Q59. Given a string s, find the length of the longest substring without repeating characters.**

String s = "abcabcbb";

int maxLength = 0;

int[] index = new int[128];

for (int j = 0, i = 0; j < s.length(); j++) {

i = Math.max(index[s.charAt(j)], i);

maxLength = Math.max(maxLength, j - i + 1);

index[s.charAt(j)] = j + 1;

}

System.out.println("Length of longest substring: " + maxLength);

**Q60. Write a Java program to check if a vowel is present in a string.**

String str = "hello";

boolean hasVowel = str.matches(".\*[AEIOUaeiou].\*");

System.out.println("Contains vowel: " + hasVowel);

**Q61. Write a Java program to check if the given number is a prime number.**

int num = 29;

boolean isPrime = true;

for (int i = 2; i <= Math.sqrt(num); i++) {

if (num % i == 0) {

isPrime = false;

break;

}

}

System.out.println("Is prime: " + isPrime);

**Q62. Write a Java program to print a Fibonacci sequence using recursion.**

public class Fibonacci {

public static void main(String[] args) {

int n = 10;

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

System.out.print(fibonacci(i) + " ");

}

}

public static int fibonacci(int n) {

if (n <= 1) return n;

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

**Q63. How do you check if a list of integers contains only odd numbers in Java?**

List<Integer> numbers = Arrays.asList(1, 3, 5, 7);

boolean allOdd = numbers.stream().allMatch(n -> n % 2 != 0);

System.out.println("All numbers are odd: " + allOdd);

**Q64. How do you check whether a string is a palindrome in Java?**

String str = "madam";

String reversed = new StringBuilder(str).reverse().toString();

boolean isPalindrome = str.equals(reversed);

System.out.println("Is palindrome: " + isPalindrome);

**Q65. How do you remove leading and trailing spaces from a string in Java?**

String str = " hello ";

String trimmed = str.trim();

System.out.println("Trimmed string: '" + trimmed + "'");

**Q66. How do you create a deadlock scenario programmatically in Java?**

public class Deadlock {

public static void main(String[] args) {

final Object lock1 = new Object();

final Object lock2 = new Object();

Thread t1 = new Thread(() -> {

synchronized (lock1) {

try { Thread.sleep(100); } catch (InterruptedException e) {}

synchronized (lock2) {}

}

});

Thread t2 = new Thread(() -> {

synchronized (lock2) {

try { Thread.sleep(100); } catch (InterruptedException e) {}

synchronized (lock1) {}

}

});

t1.start();

t2.start();

}

}

**Q67. How can you find the factorial of an integer in Java?**

public class Factorial {

public static void main(String[] args) {

int num = 5;

long factorial = 1;

for (int i = 1; i <= num; i++) {

factorial \*= i;

}

System.out.println("Factorial: " + factorial);

}

}

**Q68. How do you implement a binary search in Java?**

public class BinarySearch {

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};

int target = 5;

int index = binarySearch(arr, target);

System.out.println("Index of target: " + index);

}

public static int binarySearch(int[] arr, int target) {

int left = 0, right = arr.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) return mid;

if (arr[mid] < target) left = mid + 1;

else right = mid - 1;

}

return -1;

}

}

**Q69. Write a Java program that illustrates merge sort.**

public class MergeSort {

public static void main(String[] args) {

int[] arr = {5, 2, 9, 1, 5, 6};

mergeSort(arr, 0, arr.length - 1);

System.out.println(Arrays.toString(arr));

}

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

public static void merge(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int[] L = new int[n1];

int[] R = new int[n2];

System.arraycopy(arr, left, L, 0, n1);

System.arraycopy(arr, mid + 1, R, 0, n2);

int i = 0, j = 0, k = left;

while (i < n1 && j < n2

**Q70. Can you create a pyramid of characters in Java?**

public class CharacterPyramid {

public static void main(String[] args) {

int n = 5; // Number of rows

char ch = 'A';

for (int i = 1; i <= n; i++) {

for (int j = n; j > i; j--) {

System.out.print(" ");

}

for (int k = 1; k <= (2 \* i - 1); k++) {

System.out.print(ch);

}

System.out.println();

ch++;

}

}

}

**Q71. Write a Java program that checks if two arrays contain the same elements.**

import java.util.Arrays;

public class ArrayEquality {

public static void main(String[] args) {

int[] arr1 = {1, 2, 3, 4, 5};

int[] arr2 = {5, 4, 3, 2, 1};

Arrays.sort(arr1);

Arrays.sort(arr2);

boolean areEqual = Arrays.equals(arr1, arr2);

System.out.println("Arrays contain the same elements: " + areEqual);

}

}

**Q72. How do you get the sum of all elements in an integer array in Java?**

int[] arr = {1, 2, 3, 4, 5};

int sum = 0;

for (int num : arr) {

sum += num;

}

System.out.println("Sum of all elements: " + sum);

**Q73. How do you find the second largest number in an array in Java?**

int[] arr = {3, 5, 7, 2, 8};

int first = Integer.MIN\_VALUE, second = Integer.MIN\_VALUE;

for (int num : arr) {

if (num > first) {

second = first;

first = num;

} else if (num > second && num != first) {

second = num;

}

}

System.out.println("Second largest number: " + second);

**Q74. How do you shuffle an array in Java?**

import java.util.Collections;

import java.util.List;

import java.util.Arrays;

public class ShuffleArray {

public static void main(String[] args) {

Integer[] arr = {1, 2, 3, 4, 5};

List<Integer> list = Arrays.asList(arr);

Collections.shuffle(list);

list.toArray(arr);

System.out.println("Shuffled array: " + Arrays.toString(arr));

}

}

**Q75. How can you find a string in a text file in Java?**

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class FindStringInFile {

public static void main(String[] args) {

String filePath = "example.txt";

String searchString = "hello";

try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {

String line;

boolean found = false;

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

if (line.contains(searchString)) {

found = true;

break;

}

}

System.out.println("String found: " + found);

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Q76. How do you print a date in a specific format in Java?**

import java.text.SimpleDateFormat;

import java.util.Date;

public class DateFormatExample {

public static void main(String[] args) {

Date date = new Date();

SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

String formattedDate = formatter.format(date);

System.out.println("Formatted date: " + formattedDate);

}

}

**Q77. How do you merge two lists in Java?**

import java.util.ArrayList;

import java.util.List;

public class MergeLists {

public static void main(String[] args) {

List<Integer> list1 = new ArrayList<>(List.of(1, 2, 3));

List<Integer> list2 = new ArrayList<>(List.of(4, 5, 6));

list1.addAll(list2);

System.out.println("Merged list: " + list1);

}

}

**Q78. Write a Java program that sorts a HashMap by value.**

import java.util.\*;

public class SortHashMapByValue {

public static void main(String[] args) {

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

map.put("A", 10);

map.put("B", 30);

map.put("C", 20);

List<Map.Entry<String, Integer>> list = new ArrayList<>(map.entrySet());

list.sort(Map.Entry.comparingByValue());

LinkedHashMap<String, Integer> sortedMap = new LinkedHashMap<>();

for (Map.Entry<String, Integer> entry : list) {

sortedMap.put(entry.getKey(), entry.getValue());

}

System.out.println("Sorted HashMap: " + sortedMap);

}

}

**Q79. How do you remove all occurrences of a given character from an input string in Java?**

String str = "hello world";

char removeChar = 'o';

String result = str.replace(Character.toString(removeChar), "");

System.out.println("String after removal: " + result);

**Q80. How do you get distinct characters and their count in a string in Java?**

import java.util.HashMap;

import java.util.Map;

public class DistinctCharacters {

public static void main(String[] args) {

String str = "hello";

Map<Character, Integer> charCountMap = new HashMap<>();

for (char ch : str.toCharArray()) {

charCountMap.put(ch, charCountMap.getOrDefault(ch, 0) + 1);

}

System.out.println("Distinct characters and their count: " + charCountMap);

}

}

**Q81. Can you prove that a String object in Java is immutable programmatically?**

public class StringImmutability {

public static void main(String[] args) {

String str = "hello";

String modifiedStr = str.concat(" world");

System.out.println("Original string: " + str);

System.out.println("Modified string: " + modifiedStr);

}

}

**Q82. Can you write some code to showcase inheritance in Java?**

class Animal {

void eat() {

System.out.println("Animal is eating");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog is barking");

}

}

public class InheritanceExample {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat();

dog.bark();

}

}

**Q83. What will be the result if Java sets the main method to private?**   
If the main method is set to private, the JVM will not be able to access it, and it will result in a runtime error: Main method not found in class.

**Q84. How do you decide if an inner class is required?** An inner class is required when:

* It logically belongs to the outer class.
* It needs access to the outer class's members.
* It is not intended to be used outside the outer class.

**Q85. What are the types of inner classes in Java?** There are four types of inner classes in Java:

1. **Member Inner Class**: A class defined within another class.
2. **Static Nested Class**: A static class defined within another class.
3. **Local Inner Class**: A class defined within a method.
4. **Anonymous Inner Class**: A class without a name, defined and instantiated in a single statement.

Would you like a cheat sheet PDF or a few **interview-focused exception handling questions and answers** next?

*What is exceptional handling?*

*A,*

*Exception handling is a programming construct used to manage and respond to unexpected or exceptional conditions that can arise during the execution of a program. These exceptions typically occur when something goes wrong, such as a runtime error, invalid input, or an operation that cannot be completed successfully. Exception handling allows a program to deal with these situations gracefully, rather than crashing or terminating abruptly.*

*Key Concepts of Exception Handling*

*1. Exception:*

*• An exception is an event or condition that disrupts the normal flow of a program’s execution.*

*• Examples: Division by zero, file not found, invalid user input, or network connection errors.*

*2. Try Block:*

*• A block of code where exceptions are expected to occur.*

*• The program monitors this code for potential exceptions.*

*3. Catch Block (or Except in some languages):*

*• Code that handles specific exceptions raised in the try block.*

*• It provides a way to recover or take alternate action when an exception occurs.*

*4. Finally Block:*

*• A block that always executes after the try and catch blocks, regardless of whether an exception was raised or not.*

*• Commonly used to clean up resources like closing files or database connections.*

*5. Throw/Raise:*

*• When an exception occurs, it is “thrown” or “raised” to signal that an error has occurred.*

*6. Custom Exceptions:*

*• Developers can define their own exceptions to handle application-specific error conditions.*

*Example: Exception Handling in Java*

*public class ExceptionExample {*

*public static void main(String[] args) {*

*try {*

*int result = 10 / 0; // This will throw an ArithmeticException*

*} catch (ArithmeticException e) {*

*System.out.println("Cannot divide by zero: " + e.getMessage());*

*} finally {*

*System.out.println("Execution completed.");*

*}*

*}*

*}*

*Output:*

*Cannot divide by zero: / by zero*

*Execution completed.*

*Benefits of Exception Handling*

*1. Error Detection:*

*• Helps identify runtime errors in a controlled manner.*

*2. Improved Program Flow:*

*• Prevents abrupt program termination by allowing the program to recover or continue execution.*

*3. Readability and Maintainability:*

*• Separates error-handling logic from normal program logic.*

*4. Resource Management:*

*• Ensures resources like files, connections, etc., are properly closed or cleaned up.*

*Common Languages and Their Exception Handling Keywords*

*• Java/C#/C++: try, catch, finally, throw*

*• Python: try, except, finally, raise*

*• JavaScript: try, catch, finally, throw*

*• Ruby: begin, rescue, ensure, raise*

*Effective exception handling is critical for building robust, reliable, and user-friendly applications.*

fe operations

Explain modifiers in java?

**Modifiers in Java**

Modifiers in Java are keywords used to define the scope, access level, and behavior of classes, methods, variables, and other program elements. They are categorized into **access modifiers** and **non-access modifiers**.

**1. Access Modifiers**

Access modifiers control the visibility and accessibility of classes, methods, and variables.

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |
| **public** | ✅ | ✅ | ✅ | ✅ |
| **protected** | ✅ | ✅ | ✅ | ❌ |
| **default** | ✅ | ✅ | ❌ | ❌ |
| **private** | ✅ | ❌ | ❌ | ❌ |

**1.1 public**

* Accessible from **anywhere** in the program. **Example:**

public class PublicExample {

public int num = 10;

}

**1.2 protected**

* Accessible within the **same package** and by **subclasses**. **Example:**

class ProtectedExample {

protected int num = 10;

}

**1.3 default (Package-private)**

* Accessible only within the **same package**. **Example:**

class DefaultExample {

int num = 10; // No access modifier means default

}

**1.4 private**

* Accessible only within the **same class**. **Example:**

class PrivateExample {

private int num = 10;

}

**2. Non-Access Modifiers**

Non-access modifiers define behaviors such as **static, final, abstract**, etc.

**2.1 Static Modifier**

* Belongs to the **class**, not an instance.
* Used for variables and methods.

**Example:**

class StaticExample {

static int count = 0;

static void displayCount() {

System.out.println("Count: " + count);

}

}

**2.2 Final Modifier**

* Used to define constants, prevent method overriding, or inheritance.

**Examples:**

1. Final Variable (Constant):

final int MAX = 100;

1. Final Method:

class Parent {

final void show() {

System.out.println("Cannot override this method.");

}

}

1. Final Class:

final class FinalClass {

// This class cannot be extended.

}

**2.3 Abstract Modifier**

* Used to define abstract classes and methods.
* Abstract methods **must** be implemented in subclasses.

**Example:**

abstract class AbstractClass {

abstract void display(); // No implementation

}

class ConcreteClass extends AbstractClass {

void display() {

System.out.println("Implemented in subclass.");

}

}

**2.4 Synchronized Modifier**

* Used to control thread access to methods or blocks.

**Example:**

synchronized void syncMethod() {

// Thread-safe code

}

**2.5 Volatile Modifier**

* Ensures changes to a variable are visible to all threads.

**Example:**

volatile int sharedVar = 0;

**2.6 Transient Modifier**

* Used to skip serialization of certain variables.

**Example:**

class Example implements Serializable {

transient int transientVar; // Not serialized

}

**2.7 Native Modifier**

* Indicates a method is implemented in native code (e.g., C/C++).

**Example:**

native void nativeMethod();

**2.8 Strictfp Modifier**

* Ensures strict floating-point calculations.

**Example:**

strictfp class StrictClass {

// Floating-point operations follow IEEE 754 standard

}

**Interview Questions on Modifiers**

**Beginner Level**

1. **What is the difference between private and protected access modifiers?**
   * private: Accessible only within the same class.
   * protected: Accessible within the same package and subclasses.
2. **Can a static method access instance variables?**
   * No, static methods cannot directly access instance variables because they belong to the class, not objects.

**Intermediate Level**

1. **What happens if you declare a class as final?**
   * The class cannot be inherited.
2. **What is the use of the transient modifier?**
   * It prevents serialization of the variable.

**Advanced Level**

1. **What is the difference between synchronized and volatile modifiers?**
   * synchronized ensures thread safety by locking, while volatile ensures visibility of changes across threads.
2. **Why can’t a static method be overridden?**
   * Because static methods are associated with the class, not instances. Overriding is based on instance methods.

By understanding these modifiers, you can define clear scopes and behaviors for your Java programs effectively. Let me know if you need examples for any specific modifier!

Explain final finally finalize?

**Final, Finally, and Finalize in Java**

These three terms may sound similar, but they serve different purposes in Java. Let's break them down:

**1. final**

**Purpose:**

The final keyword is used to define constants, prevent inheritance, or prohibit method overriding.

**Use Cases:**

1. **Final Variables:**
   * The value of a final variable cannot be changed after initialization.
   * Must be initialized either at the time of declaration or in the constructor.

**Example:**

java

CopyEdit

public class FinalVariableExample {

final int MAX\_VALUE = 100; // Constant

void display() {

// MAX\_VALUE = 200; // Error: Cannot reassign a final variable

System.out.println("MAX\_VALUE: " + MAX\_VALUE);

}

}

1. **Final Methods:**
   * A final method cannot be overridden by subclasses.

**Example:**

java

CopyEdit

class Parent {

final void display() {

System.out.println("This method cannot be overridden.");

}

}

class Child extends Parent {

// void display() {} // Error: Cannot override final method

}

1. **Final Classes:**
   * A final class cannot be extended (i.e., no subclasses can be created).

**Example:**

java

CopyEdit

final class FinalClass {

void display() {

System.out.println("This class cannot be inherited.");

}

}

// class SubClass extends FinalClass {} // Error: Cannot inherit from a final class

**2. finally**

**Purpose:**

The finally block is used in exception handling to execute a block of code regardless of whether an exception is thrown or not.

**Key Points:**

* It is **optional** and must be used with a try block.
* Commonly used for **cleanup operations**, like closing resources.

**Example:**

java

CopyEdit

public class FinallyExample {

public static void main(String[] args) {

try {

int result = 10 / 0; // Exception occurs here

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e.getMessage());

} finally {

System.out.println("Finally block executed.");

}

}

}

**Output:**

vbnet

CopyEdit

Exception caught: / by zero

Finally block executed.

**When Does finally NOT Execute?**

* If the JVM terminates abruptly (e.g., using System.exit(0)).

**Example:**

java

CopyEdit

try {

System.exit(0);

} finally {

System.out.println("This will not execute.");

}

**3. finalize**

**Purpose:**

The finalize method is called by the Garbage Collector (GC) before an object is destroyed to allow cleanup operations.

**Key Points:**

* Defined in the Object class.
* Rarely used in modern Java due to the unpredictability of when garbage collection occurs.

**Syntax:**

java

CopyEdit

@Override

protected void finalize() throws Throwable {

System.out.println("Object is being garbage collected.");

}

**Example:**

java

CopyEdit

public class FinalizeExample {

public static void main(String[] args) {

FinalizeExample obj = new FinalizeExample();

obj = null; // Make the object eligible for garbage collection

System.gc(); // Request garbage collection

}

@Override

protected void finalize() throws Throwable {

System.out.println("Finalize method called before object destruction.");

}

}

**Output (example):**

sql

CopyEdit

Finalize method called before object destruction.

Explain super keyword?

A,

**When and Why Do We Use super?**

1. **Access Parent Class Members:**
   * In cases where a subclass has variables or methods with the same name as those in the parent class, super is used to explicitly refer to the parent class members, avoiding ambiguity.
2. **Call Parent Class Constructors:**
   * When creating an object of a subclass, the constructor of the parent class is called first. super is used to explicitly invoke a specific constructor in the parent class.
   * This is useful when the parent class does not have a no-argument constructor or when specific parameters need to be passed to the parent class.
3. **Override Parent Class Methods:**
   * If a method in the subclass overrides a method from the parent class, super can be used within the overriding method to call the original method from the parent class.

Ans,

class Parent {

int num = 100;

}

class Child extends Parent {

int num = 200;

void display() {

System.out.println("Child class variable: " + num); // Refers to Child class variable

System.out.println("Parent class variable: " + super.num); // Refers to Parent class variable

}

}

public class SuperExample1 {

public static void main(String[] args) {

Child obj = new Child();

obj.display();

}

}