Oops concepts.

Modifies:access, non access(static,final).

constructors

Oops concepts  
**Object-Oriented Programming (OOP) Concepts Explained with Answers to Interview Questions**

**OOP Concepts**

**1. Encapsulation**

Encapsulation bundles data (variables) and methods (functions) that operate on the data into a single unit (class) and restricts direct access to data members using access modifiers.

**Example**:

public class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

public double getBalance() {

return balance;

}

}

**Q&A**:  
**Q: What is encapsulation? How is it implemented?**  
**A:** Encapsulation is the mechanism of wrapping data and methods in a single unit, ensuring controlled access using access modifiers like private, public, or protected. It is implemented by making class variables private and providing public getter and setter methods for controlled access.

**2. Inheritance**

Inheritance allows a class (child) to acquire properties and behaviors from another class (parent), promoting code reusability.

**Example**:

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited method

dog.bark(); // Method specific to Dog

}

}

**Q&A**:  
**Q: What is inheritance? What are its types?**  
**A:** Inheritance is the process by which one class acquires the properties and methods of another class.

* **Single inheritance**: One class inherits from another.
* **Multilevel inheritance**: A class inherits from a derived class.
* **Hierarchical inheritance**: Multiple classes inherit from a single base class.  
  (Note: Java does not support multiple inheritance with classes to avoid ambiguity; it uses interfaces instead.)

**3. Polymorphism**

Polymorphism means "many forms" and allows methods to perform differently based on the object or input.

**Method Overloading (Compile-Time Polymorphism):**

class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

**Method Overriding (Run-Time Polymorphism):**

class Animal {

void sound() {

System.out.println("This animal makes a sound.");

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("The cat meows.");

}

}

**Q&A**:  
**Q: What is polymorphism? How is it achieved?**  
**A:** Polymorphism allows the same method or operator to behave differently. It is achieved through:

* **Compile-Time Polymorphism**: Using method overloading (methods with the same name but different parameters).
* **Run-Time Polymorphism**: Using method overriding (subclass method overrides the superclass method).

**4. Abstraction**

Abstraction hides implementation details and exposes only the essential features.

**Abstract Class Example:**

abstract class Shape {

abstract void draw(); // Abstract method

}

class Circle extends Shape {

void draw() {

System.out.println("Drawing a circle.");

}

}

**Interface Example:**

interface Vehicle {

void start();

}

class Car implements Vehicle {

public void start() {

System.out.println("Car starts with a key.");

}

}

Explain about modifiers ?

Absolutely! Let's go in-depth on **Java Modifiers** and then cover the most common **interview questions** specifically related to modifiers.

**✅ Java Modifiers Explained**

Java modifiers are **keywords** used to define the **scope**, **behavior**, and **access level** of classes, methods, constructors, and variables.

**🔹 1. Access Modifiers – Control Visibility**

| **Modifier** | **Class** | **Package** | **Subclass (Different Pkg)** | **World** |
| --- | --- | --- | --- | --- |
| public | ✔ | ✔ | ✔ | ✔ |
| protected | ✔ | ✔ | ✔ | ✖ |
| *default* | ✔ | ✔ | ✖ | ✖ |
| private | ✔ | ✖ | ✖ | ✖ |

**📌 Usage**

* public: Accessible from anywhere.
* protected: Accessible within package + subclasses.
* *(no modifier)*: Accessible only within the package.
* private: Accessible only within the class.

**🔹 2. Non-Access Modifiers**

**➤ For Classes**

* final: Class cannot be inherited.
* abstract: Class cannot be instantiated (may contain abstract methods).
* strictfp: Restricts floating-point precision consistency across platforms.

**➤ For Methods**

* final: Cannot be overridden by subclasses.
* abstract: Must be implemented by a subclass.
* static: Belongs to the class, not the instance.
* synchronized: Used in multithreading to lock access.
* native: Declares that the method is implemented in native code (like C/C++).
* strictfp: Ensures consistent floating-point calculations.

**➤ For Variables**

* final: Value cannot be changed once assigned.
* static: Shared among all instances of a class.
* transient: Excluded during serialization.
* volatile: Ensures changes made by one thread are visible to others.

.

**✅ Java Modifiers Cheat Sheet**

**🔹 Access Modifiers**

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

* **public**: Accessible everywhere
* **protected**: Accessible in same package + subclass (even in other packages)
* **default**: (No modifier) Only within same package
* **private**: Only within the class

**🔹 Non-Access Modifiers**

**➤ For Classes**

| **Modifier** | **Meaning** |
| --- | --- |
| final | Class can't be extended (no subclassing) |
| abstract | Class can't be instantiated, may have abstract methods |
| strictfp | Enforces floating-point consistency |

**➤ For Methods**

| **Modifier** | **Meaning** |
| --- | --- |
| final | Can't be overridden |
| abstract | Must be implemented in subclass |
| static | Belongs to class, not instance |
| synchronized | Thread-safe access |
| native | Implemented in non-Java (native) code |
| strictfp | Applies IEEE floating-point precision |

**➤ For Variables**

| **Modifier** | **Meaning** |
| --- | --- |
| final | Constant – value can't change |
| static | Shared across all instances |
| transient | Not serialized |
| volatile | Always read from main memory (used in multithreading) |

**Modifiers**

1. **What are access modifiers in Java, and how do they affect the visibility of a class, method, or variable?**
   * Access modifiers in Java (public, protected, private, and default) control the visibility of classes, methods, and variables. public makes them accessible from any other class, protected allows access within the same package and subclasses, private restricts access to the defining class, and default (no modifier) allows access within the same package.
2. **Explain the difference between public, protected, private, and default access modifiers.**
   * public: Accessible from any class.
   * protected: Accessible within the same package and subclasses.
   * private: Accessible only within the defining class.
   * Default (no modifier): Accessible within the same package.
3. **What is the purpose of the final modifier in Java? Can you give an example of its usage?**
   * The final modifier is used to define constants, prevent method overriding, and inheritance. Example: final int MAX\_VALUE = 100;.
4. **How does the static modifier work, and when would you use it?**
   * The static modifier indicates that a member belongs to the class rather than instances of the class. It is used for class-level variables and methods. Example: static int count;.
5. **What is the abstract modifier, and how is it used in classes and methods?**
   * The abstract modifier is used to declare abstract classes and methods that must be implemented by subclasses. Example: abstract class Animal { abstract void sound(); }.
6. **Can you explain the synchronized modifier and its role in thread safety?**
   * The synchronized modifier ensures that a method or block of code is accessed by only one thread at a time, preventing race conditions. Example: synchronized void increment() { count++; }.
7. **What is the transient modifier, and when is it used?**
   * The transient modifier is used to indicate that a variable should not be serialized. Example: transient int tempData;.
8. **Describe the volatile modifier and its significance in concurrent programming.**
   * The volatile modifier ensures that a variable's value is always read from the main memory, providing visibility guarantees in concurrent programming. Example: volatile boolean flag;.
9. **How does the native modifier work, and when would you use it?**
   * The native modifier is used to declare methods that are implemented in native code using JNI (Java Native Interface). Example: native void print();.
10. **What is the strictfp modifier, and why would you use it in Java?**
    * The strictfp modifier ensures that floating-point calculations follow IEEE 754 standards for portability. Example: strictfp class Calculator { ... }.

**Constructors**

1. **What is a constructor in Java, and how does it differ from a method?**
   * A constructor is a special method used to initialize objects. It differs from a method in that it has no return type and is called automatically when an object is created.
2. **Explain the concept of constructor overloading with an example.**
   * Constructor overloading allows multiple constructors with different parameters in a class. Example:
3. **What is the purpose of the default constructor, and when is it provided by the compiler?**
   * The default constructor is provided by the compiler if no constructors are defined. It initializes the object with default values.
4. **How can you prevent a class from being instantiated?**
   * You can prevent a class from being instantiated by making its constructor private. Example:
5. **What is a copy constructor, and how is it implemented in Java?**
   * A copy constructor creates a new object as a copy of an existing object. Example:
6. **Can you explain the use of the this keyword in constructors?**
   * The this keyword refers to the current object and is used to differentiate instance variables from parameters. Example:
7. **What is the role of the super keyword in constructors?**
   * The super keyword is used to call the superclass constructor. Example:
8. **How can you call one constructor from another within the same class?**
   * You can call one constructor from another using this(). Example:
9. **What are the differences between a constructor and a static block?**
   * A constructor initializes an instance of a class, while a static block initializes static variables and is executed when the class is loaded.
10. **How does constructor chaining work, and why is it useful?**
    * Constructor chaining involves calling one constructor from another to reuse code. It is useful for initializing objects in a consistent manner. Example: