OVERVIEW

Object-Oriented Programming (OOP) is a key concept that you're expected to know if you come from a technical background. To help you nail your next interview, we've compiled the top 100+ OOPs interview questions and answers to help you master the essential concepts and techniques.

Note : We have compiled all OOPs Interview Questions for you in a template format. Check it out now!

OOPs Interview Questions for Freshers

Here are the OOPs interview questions and answers for freshers that cover the fundamental concepts of Object-Oriented Programming, such as inheritance, polymorphism, encapsulation, and abstraction.

This is one of the most commonly asked OOPs interview questions. Object-Oriented Programming is a programming paradigm that uses objects instead of functions and procedures. These objects are grouped into classes. OOP incorporates real-world entities like inheritance, polymorphism, and encapsulation into programming, allowing for the integration of data and methods.

2. Why Use OOPs?

OOPs need to be used for several reasons:

- OOP organizes code into objects, making complex problems easier to understand and solve.
- Properties and methods from parent classes are inherited by child classes, promoting efficient code reuse.
- By using inheritance and creating reusable objects, OOP minimizes code duplication.
- OOP bundles data and the functions that manipulate it within a single unit or object.
- Access modifiers in OOP control the visibility of internal object details, enhancing security and reducing system complexity.
- OOP allows breaking down complex problems into smaller, manageable subproblems represented by objects.
- It allows objects from different classes to be treated as objects of a common superclass, increasing code flexibility and extensibility.

3. What Are Some Major Object-Oriented Programming Languages?

This is a frequently asked OOP interview question. Some major Object-Oriented Programming languages include:

- Java
- C++
- C#
- Python
- Ruby
- JavaScript
- Swift
- PHP
- Objective-C

4. What Are the Main Features of OOPs?

The following are the main features of OOPs:

- **Classes:** Classes in OOP are essential components, acting as templates for objects. They define the data structure and behavior of objects, encapsulating attributes and methods to provide a modular and organized design framework.
- **Objects:** In OOP, objects are core elements that represent real-world entities with distinct attributes and behaviors. As instances of classes, they encapsulate data and methods, enhancing code modularity and organization. Objects communicate via well-defined interfaces, supporting structured design principles.
- **Inheritance:** Inheritance enables a class to inherit fields and methods from another class, fostering code reuse and creating a hierarchical class structure.
- **Abstraction:** In OOP, abstraction simplifies complex systems by creating classes modeled on real-world entities, emphasizing essential features and concealing unnecessary details. This approach makes clear and concise designs, allowing developers to focus on core attributes and behaviors without the intricacies of implementation.
- **Encapsulation:** Encapsulation combines data and methods that operate on that data into a single unit called a class. It also restricts direct access to some of an object's components.
- **Polymorphism:** Polymorphism, a key feature of OOP, allows objects from different classes to be treated as objects of a common base class.

5. What Is an Object?

An object in Object-Oriented Programming is a fundamental unit representing real-world entities. It is an instance of a class, with memory allocated only upon instantiation (object creation). Objects possess identity, state, and behavior, containing data and code to manipulate it. Interaction between objects doesn't require knowledge of each other's data or code specifics, just the types of messages accepted and responses provided.

6. What Is a Class?

A class is a user-defined data type that consists of data members and member functions accessible through an instance of the class. It represents the common properties or methods of all objects of a specific type, functioning as a blueprint for an object.

7. What Is the Difference Between a Class and a Structure?

This is one of the most commonly asked OOPs interview questions. Here are the differences between a class and a structure:

Feature	Class	Structure
Access Modifiers	Supports public, private, and protected access modifiers.	Typically only supports public access modifiers (implementation-dependent).
Inheritance	Supports inheritance (single or multiple, depending on the language).	Does not support inheritance (except in C++, where it supports single inheritance).
Constructor/Destructor	Has constructors and destructors.	Does not have constructors or destructors.
Method Overloading	Supports method overloading.	Supports method overloading (implementation-dependent).
Method Overriding	Supports method overriding (in case of inheritance).	Does not support method overriding.
Default Members	Members are private by default (in most languages).	Members are public by default.
Size	Can have a size larger than the sum of its members (due to additional metadata).	Size is equal to the sum of its members' sizes.
Memory Allocation	Objects are typically allocated on the heap.	Structures are typically allocated on the stack or embedded in other data structures.

Recommended Usage	Used for complex data types and to model real-world entities.	Used for simple, lightweight data types and grouping-related data.

8. What Are the Differences Between Class and Object?

The primary differences between a class and an object in Object-Oriented Programming are:

Aspect	Class	Object
Definition	A class is a template or blueprint that defines the attributes and actions of objects.	An object is a specific instance of a class created from the class blueprint.
Memory Allocation	A class itself does not occupy any memory space.	Objects occupy memory space when they are created.
Creation	Classes are created by the programmer during the design phase.	Objects are created from classes at runtime, typically using constructors or object creation expressions.
Properties and Methods	A class defines the properties and methods that objects will have.	An object has its own copies of the properties and can access and modify them. It also has access to the methods defined in the class.
Instantiation	A class cannot be instantiated directly.	Objects can be instantiated from a class using the new keyword or object creation expressions.
Purpose	A class serves as a blueprint or template.	Objects are the actual entities that exist and perform tasks during runtime.

Access Modifiers	Classes can have access modifiers (public, private, protected) that control access to their members.	Objects do not have access modifiers; they access members based on the access modifiers defined in the class.
Inheritance	Classes can inherit properties and methods from other classes through inheritance.	Objects cannot inherit directly from other objects but from the class they are instantiated from.
Polymorphism	Classes support polymorphism, allowing objects of different classes to be treated as objects of a common superclass.	Objects can exhibit polymorphic behavior based on the class they are instantiated from.

9. What Are Some Advantages of Using OOPs?

Some of the primary advantages of OOPs are as follows:

- **Code Reusability:** Utilizes class-based architecture for modular design, enabling instantiation of objects from reusable class templates. This promotes code efficiency through multiple instantiations.
- **Data Redundancy Management:** Implements controlled redundancy for data integrity, facilitates data consistency across multiple instances, and utilizes inheritance for shared class definitions.
- **Enhanced Security:** Implements encapsulation for data protection, utilizes access modifiers to control the visibility of class members and employs abstraction to expose only necessary interfaces.
- **Improved Design Methodology:** Facilitates comprehensive design phase, supports scalable architecture for complex systems, and provides flexibility in handling system growth beyond the initial scope.
- **Efficient Debugging:** Utilizes self-contained objects for localized error isolation, implements exception handling for robust error management, and reduces code duplication, minimizing potential error sources.
- **Polymorphism:** Enables method overloading and overriding, supports runtime polymorphism through interface implementation, and enhances code flexibility and

extensibility.

10. What Is the Difference Between Object-Oriented and Structured Programming?

Here are the main differences between them:

Aspect	Object-Oriented Programming	Structured Programming
Approach	OOP focuses on creating objects that contain data and behavior. The program is organized around objects and their interactions.	Structured Programming focuses on writing a set of instructions or functions to perform tasks. The program is organized around procedures or functions.
Data and Functions	In OOP, data and functions are combined into objects. Data is encapsulated within objects, and functions (methods) operate on that data.	In Structured Programming, data and functions are separate entities. Functions operate on data passed as arguments.
Code Organization	Code is organized around classes and objects, promoting code reuse and modularity.	Code is organized around procedures or functions, often leading to monolithic and less modular code.

Data Abstraction	OOP supports data abstraction and encapsulation, where implementation details are hidden from the user.	Structured Programming has limited support for data abstraction and encapsulation.
Inheritance	OOP supports inheritance.	Structured Programming does not support inheritance.
Polymorphism	OOP supports polymorphism.	Structured Programming does not support polymorphism.
Code Reuse	OOP promotes code reuse through the inheritance and composition of objects.	Structured Programming has limited code reuse capabilities, often achieved through function libraries or modules.

11. What Other Paradigms of Programming Exist Besides OOPs?

This is one of the most commonly asked OOPs interview questions. Programming paradigms categorize programming languages based on their core characteristics.

There are two main types:

- Imperative Programming Paradigm
- Declarative Programming Paradigm

These paradigms can be further subdivided:

- **Imperative Programming Paradigm:** This approach specifies how to execute program logic and defines a sequence of commands to change program state. It includes:
  - **Procedural Programming Paradigm:** This method defines a set of computational steps to be carried out and executed in order from top to bottom.
  - **Object-Oriented Programming:** This organizes software design around data or objects rather than functions and logic.
  - **Parallel Programming:** This focuses on breaking down tasks into smaller components that can be processed concurrently.

- **Declarative Programming Paradigm:** This approach emphasizes what the program should accomplish without explicitly specifying control flow. It includes:

- **Logical Programming Paradigm:** Based on formal logic, this uses a set of statements expressing facts and rules to solve problems.
- **Functional Programming Paradigm:** This builds programs by composing and applying functions, avoiding changing-state and mutable data.
- **Database Programming Paradigm:** This model manages data organized into fields, records, and files.

12. What Are the Differences Between Procedural and Object-Oriented Programming?

The following are the differences between Procedural Programming and Object-Oriented Programming:

Aspect	Procedural Programming	Object-Oriented Programming
Data	Separate from procedures	Encapsulated within objects
Program structure	Top-down approach	Divided into objects
Data access	Usually global	Private (encapsulated)
Code reusability	Limited, mainly through functions	High, through inheritance
Data security	Less secure	More secure due to encapsulation
Overloading	Generally not supported	Supports function and operator overloading
Complexity	Simpler for small programs	Better for managing complex programs

Modularity	Less modular	Highly modular
Abstraction level	Low level of abstraction	High level of abstraction
Inheritance	Not supported	Supported
Polymorphism	Not supported	Supported
Examples	C, Pascal	Java, C++, Python, C#

13. What Are Access Specifiers, and When Should We Use These?

This is a popular OOP interview question. As the name indicates, access specifiers are special types of keywords used to specify or control the accessibility of entities such as classes and methods.

The three primary access specifiers are:

- Public
- Private
- Protected

When to use each:

- **Public**: Use when you want class members to be accessible from anywhere in the program. They are typically used for methods that need to be called by external code or for data that should be freely accessible.
- **Private**: Use when you want to restrict access to class members only within the same class. Helps in hiding implementation details and maintaining data integrity. Useful for internal helper methods or variables that should not be directly manipulated from outside the class.
- **Protected**: Use when you want to allow access to class members within the same class and its subclasses. Useful when implementing inheritance and you want to share some internal functionality with derived classes.

14. What Is Encapsulation?

Think of encapsulation like a capsule. You see a simple shell on the outside, but inside, it contains everything needed to do its job. In programming, encapsulation works similarly:

- It bundles related data and functions together into a single unit (like a class).
- It hides the complex internal functioning and shows only what's necessary to use it.

15. What Is Abstraction?

This is one of the frequently encountered OOPs interview questions. A typical OOPs interview question, this one is often asked. Abstraction in Object-Oriented Programming hides non-essential information and shows only what is necessary to users. This is key to representing real-world objects simply for easy user interaction.

There are two types of abstractions in Object-Oriented Programming:

- **Data Abstraction**: Data abstraction is the most simple implementation of abstraction. When dealing with many complex processes, we hide unnecessary data and display only the information necessary for the users.
- **Process Abstraction**: When we hide the internal implementation and do not disclose all the details about a method or function to the users, this is known as process abstraction. Instead of dealing with data, we deal with their process in this type of abstraction.

16. What Are Some Recommended Best Practices for Applying Abstraction in Your Code?

This is a common OOPs interview question that is regularly asked. Here are a few best practices for using abstraction in your code:

- Creating well-defined interfaces or abstract classes.
- Keeping designs simple and avoiding over-engineering.

- Prioritizing composition over inheritance.

17. What Is Polymorphism?

This is one of the most commonly asked OOPs interview questions. Polymorphism is a core concept in Object-Oriented Programming that allows objects of various types to be treated in a uniform way. It makes code more flexible and reusable. It allows for writing methods that can work with objects of multiple types as long as they share a common superclass or interface.

For example, you could have a method that processes "Shape" objects, and it would work correctly whether you pass it a "Circle," "Square," or "Triangle" object, as long as they're all subclasses of "Shape."

18. What Are the Different Types of Polymorphism?

There are two main types of polymorphism in Object-Oriented Programming:

- **Compile-Time Polymorphism (Static Polymorphism)**
  - Also known as method overloading.
  - Resolved during compile time.
  - Occurs when multiple methods in the same class have the same name but different parameters.
  - The compiler specifies which method to call based on the method signature.
- **Runtime Polymorphism (Dynamic Polymorphism)**:
  - Also known as method overriding.
  - Resolved during runtime.
  - Occurs when a subclass gives a specific implementation for a method already defined in its superclass.
  - The JVM determines which method to call based on the object's actual type at runtime.

19. What Is Method Overloading?

When a class has multiple methods with the same name but different parameters, it is called Method Overloading.

Characteristics of method overloading:

- **Same Method Name:** All overloaded methods share the same name within a class.
- **Different Parameters:** The methods must differ in the number, type, or order of parameters.
- **Return Type:** The return type can be different, but this alone is not enough to overload a method.
- **Compile-Time Polymorphism:** Method overloading in Java is resolved at compile time, not at runtime.

Let's understand this with an example in Java:

```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

20. What Is Method Overriding?

This is a standard OOPs interview question. When a subclass (child class) in Java implements a method that has the same name and parameters as a method in its parent class, this is called method overriding.

Characteristics of method overriding:

- **Same Method Signature:** The method in the subclass that overrides a method from the superclass must have the same name, return type, and parameter list as the method in the superclass.
- **Runtime Polymorphism:** The decision about which method to call is made at runtime, not at compile time.
- **Access Modifiers:** The overriding method must not have a stricter access level than the overridden method.

Let's understand this with an example in Java:

```
class Shape {
    double getArea() {
        return 0;
    }
}

class Circle extends Shape {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double getArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double width, height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    double getArea() {
```

```
        return width * height;
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape1 = new Circle(5);
        Shape shape2 = new Rectangle(4, 6);

        System.out.println("Circle area: " + shape1.getArea());
        System.out.println("Rectangle area: " + shape2.getArea());
    }
}
```

21. What Is the Difference Between Overloading and Overriding?

This is one of the most commonly asked OOPs interview questions. Here's the difference between overloading and overriding:

Aspect	Overloading	Overriding
Definition	Methods with identical names and differing parameters.	Implementing a method in a subclass that is already defined in its superclass.
Purpose	Provides method implementation flexibility.	Allows subclasses to provide specific implementation.
Class	Occurs within the same class or between parent and child classes.	Occurs between superclass and subclass.
Method Signature	Must have different parameter types or a number of parameters.	Must have the same method signature (name and parameters).
Return Type	Can be different	Must be the same or covariant

Access Modifier	Can be different	Cannot be more restrictive in the subclass
Static/Non-static	Can be either	Must be non-static
Binding	Compile-time (static) binding	Runtime (dynamic) binding

22. What Is the Difference Between Compile-Time Polymorphism and Runtime Polymorphism?

Here's the difference between compile-time polymorphism and runtime polymorphism:

Aspect	Compile-Time Polymorphism	Runtime Polymorphism
Resolution Time	Compile time	Runtime
Also Known As	Static binding, early binding	Dynamic binding, late binding
Implementation	Method overloading, operator overloading	Method overriding, Inheritance
Flexibility	Less flexible	More flexible
Performance	Generally faster	Slightly slower due to runtime resolution
Decision Making	Before program execution	During program execution
Type Checking	Done by compiler	Done at runtime

23. Is It Possible to Achieve Runtime Polymorphism Through Data Members in Java?

No, runtime polymorphism cannot be achieved by data members in languages like Java or C++ because:

- Runtime polymorphism is achieved through method overriding, not data members (variables).
- Methods can be overridden, but data members cannot be overridden.
- When accessing a data member using a reference variable of the parent class that refers to a child class object, the data member of the parent class will always be accessed.
- This is because data members are not overridden, so runtime polymorphism doesn't apply to them.
- What occurs with data members is actually called "hiding" rather than overriding.

24. What Is Inheritance?

Inheritance in Object-Oriented Programming describes the ability of a class to inherit features from a superclass.

Syntax:

```
class SubClass extends SuperClass {
// SubClass members
}
```

Key elements of the syntax:

- Use the keyword class to declare a new class.
- After the class name, use the keyword extends.
- Following extends, specify the name of the superclass (parent class).
- The subclass body is enclosed in curly braces .

25. What Is a Superclass?

The superclass is the class that provides features inherited by a subclass, also known as the base class or parent class.

26. What Is a Subclass?

A subclass is a class that inherits all members (fields, methods, and nested classes) from another class, also known as a derived class, child class, or extended class.

27. What Different Types of Inheritance Are There?

This is a common OOPs interview question that is regularly asked. In Object-Oriented Programming, inheritance is categorized by the relationship between the derived class and its superclass.

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

28. Are There Any Limitations on Inheritance?

This is one of the frequently encountered OOPs interview questions. Despite its strength in Object-Oriented Programming, inheritance does have its set of limitations:

- **Performance Overhead:** Inherited methods typically execute more slowly than regular functions due to the additional layer of indirection.
- **Potential for Misuse:** Incorrect application of inheritance can result in flawed or inappropriate solutions to programming problems.
- **Inefficient Resource Utilization:** Base classes may contain data members that remain unused in derived classes, potentially leading to unnecessary memory consumption.
- **Increased Interdependence:** Inheritance creates a tighter coupling between parent and child classes. Modifications to the base class can have far-reaching effects on all its descendants.

29. Why Is Inheritance Used in Java?

Inheritance is an essential feature in Object-Oriented Programming languages like Java and C++.

Inheritance is like a parent passing traits to their child. In programming, it lets one class (the child) get all the characteristics and abilities of another class (the parent). This means the child class can use the parent's data and functions without rewriting them.

The main idea is to make new classes based on existing ones. The new class (child) gets everything from the old class (parent) and can add its unique features.

With inheritance, we'd save time writing the same code repeatedly for similar classes. It's like reinventing the wheel each time instead of just using a wheel that already exists.

30. How to Implement Inheritance in Java?

Inheritance can be implemented by using:

- ***extends:*** Used to create inheritance between two classes.
- ***implements:*** Used to create inheritance between a class and an interface.

31. Is It Possible for a Class to Extend Itself?

It's not possible for a class to extend itself.

32. What Is Composition?

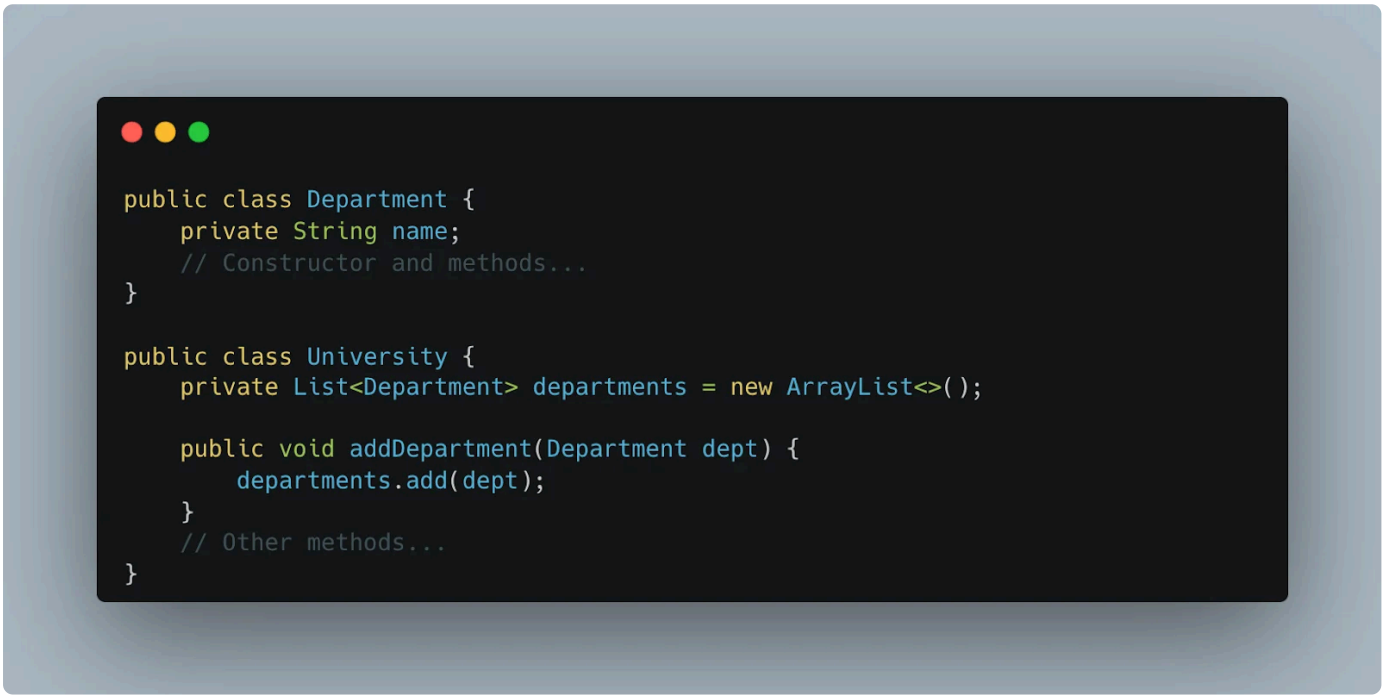
Composition is a design approach in Object-Oriented Programming to establish a has-a relationship between objects. In Java, this is done by including instance variables of other objects within a class.

33. What Is the Difference Between Aggregation and Composition?

Aggregation: Aggregation represents a weak association between two entities where one entity can contain or use another, but the contained entity can exist independently. It's characterized by:

- A relationship where the contained entity can exist without the container.
- The ability for the contained entity to be associated with multiple containers.
- No direct control of the contained entity's lifespan by the container.
- Implementation typically using references or pointers to existing objects.

Example in Java:



```java
public class Department {
    private String name;
    // Constructor and methods...
}

public class University {
    private List<Department> departments = new ArrayList<>();

    public void addDepartment(Department dept) {
        departments.add(dept);
    }
    // Other methods...
}
```

Composition: Composition denotes a strong, whole-part relationship between two entities where the part cannot exist without the whole.

Key characteristics include:

- The part is an integral component of the whole and cannot exist independently.
- The whole is responsible for the creation and destruction of its parts.
- A part typically belongs to only one whole at a time.
- Implementation often involves creating part objects within the whole's constructor.

Example in Java:

```java
public class Engine {
    private int horsepower;
    // Constructor and methods...
}

public class Car {
    private Engine engine;

    public Car() {
        this.engine = new Engine(200);
    }
    // Other methods...
}
```

The major differences include:

- **Entity Independence:** Aggregated entities can exist independently; composed entities cannot.
- **Relationship Exclusivity:** Aggregated entities can be shared; composed entities typically belong to one whole.
- **Lifespan Management:** Aggregating objects don't control the lifespan of aggregated entities; composing objects do control the lifespan of their parts.
- **Coupling:** Aggregation results in loose coupling, while composition creates tighter coupling between entities.

34. What Is the Difference Between Inheritance and Polymorphism?

Following are the differences between inheritance and polymorphism:

Aspect	Inheritance	Polymorphism
Definition	A mechanism where a class inherits properties and behaviors from another class.	Objects of different types that are treated as objects of a common base type.

Purpose	Code reuse and establishing a hierarchical relationship between classes.	Flexibility in using different object types through a common interface.
Relationship	Establishes an "is-a" relationship between classes.	Enables "one interface, multiple implementations".
Implementation	Achieved using the extends keyword (in many languages).	Achieved through method overriding and interfaces/abstract classes.
Types	Single inheritance, multiple inheritance (in some languages).	Runtime polymorphism, compile-time polymorphism.
Coupling	Creates tight coupling between parent and child classes.	Promotes loose coupling through abstraction.
Focus	Emphasizes sharing of code and behavior.	Emphasizes interchangeability of objects.

35. Can You Call the Base Class Method Without Creating an Instance?

Often asked, this OOPs interview question is common. In Object-Oriented Programming languages like C#, instance methods of a base class are tied to specific objects and cannot be invoked without an instance. This means you typically need to create an object of either the base class or a derived class to call these methods. However, static methods in a base class operate at the class level rather than the instance level. These can be called directly using the class name, without the need to instantiate an object.

36. What Is a Constructor?

Constructors are special methods or functions in Object-Oriented Programming used to initialize class objects, defining their initial state and creation method. They typically share the same name as their class.

37. How Many Types of Constructors Are Used in Java?

Constructors in Java are classified into three types:

- No-Arg constructor
- Parameterized constructor
- Default constructor

38. What Is the Purpose of a Default Constructor?

The purpose of a default constructor is to:

- Initialize an object to a default state when no arguments are provided during object creation.
- Ensure that an object can be created without explicitly specifying initial values.
- Provide a fallback initialization method if no other constructors are called.
- Allow the class to be instantiated in arrays or collections that require a no-argument constructor.
- Enable derived classes to have a default superclass constructor to call.

39. Does the Constructor Return Any Value?

Constructors are special functions in Object-Oriented Programming that initialize newly created objects. Unlike regular methods, constructors don't have a return type and are implicitly called when an object is instantiated. Their primary purpose is to set up the initial state of an object by initializing instance variables.

Constructors share the same name as their class and can be overloaded to accept different parameters. While they resemble methods in some ways, constructors are distinct in that they can't be directly called and aren't part of the object's interface.

Many languages provide a default no-argument constructor if none is explicitly defined, and constructors can often chain to other constructors or superclass constructors to build complex initialization sequences.

40. Can We Overload the Constructor in a Class?

Yes, we can overload constructors in a class. Constructor overloading indeed works similarly to function overloading. The key aspects are:

- **Same Name:** All constructors have the same name as the class.
- **Different Parameters:** Overloaded constructors differ in their parameter lists.
- **Automatic Selection:** The appropriate constructor is called based on the arguments provided during object creation.

Here's an example demonstrating constructor overloading in Java:

```java
public class Rectangle {
    private int length;
    private int width;

    // No-argument constructor
    public Rectangle() {
        length = 0;
        width = 0;
    }

    // Constructor with one argument
    public Rectangle(int side) {
        length = side;
        width = side;
    }

    // Constructor with two arguments
    public Rectangle(int l, int w) {
        length = l;
        width = w;
    }

    // Method to display dimensions
    public void displayDimensions() {
        System.out.println("Length: " + length + ", Width: " + width);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();         // Calls no-argument constructor
        Rectangle r2 = new Rectangle(5);        // Calls one-argument constructor
        Rectangle r3 = new Rectangle(4, 6);     // Calls two-argument constructor

        r1.displayDimensions();
        r2.displayDimensions();
        r3.displayDimensions();
    }
}
```

This example shows three overloaded constructors for the Rectangle class:

- A no-argument constructor that sets both dimensions to 0.
- A one-argument constructor that creates a square.
- A two-argument constructor that sets length and width separately.

The appropriate constructor is automatically called based on the arguments provided when creating new Rectangle objects.

41. What Is Constructor Chaining?

Constructor chaining in Object-Oriented Programming is the method of using one constructor to invoke another within the same class. This approach helps simplify the program by minimizing redundant code.

Constructor chaining can be achieved in two ways:

- **Within the Same Class:** Use this for chaining constructors within the same class.
- **From the Base Class:** Use superfor chaining constructors from different classes (parent and child).

42. What Are the Differences Between the Constructor and the Method in Java?

Constructors and methods are fundamental components of Object-Oriented Programming, but they serve different purposes and have distinct characteristics.

- **Invocation:**

  Constructor: A constructor is invoked implicitly when an object is created using the new keyword.

  ```
  Person person = new Person();
  ```

  Method: A method is invoked explicitly by the programmer through method calls.

  ```
  person.setName("peter");
  ```

- **Naming:**

  Constructor: The name of a constructor must be the same as the name of the class in which it is defined.

  Method: A method can have any name, except it cannot be the same as the class name if it has a return type.

- **Return Type:**

Constructor: Constructors do not have a return type, not even void. The object being constructed is implicitly returned.

Method: Methods must have a return type, which can be a primitive type, an object type, or void if no value is returned.

- **Inheritance:**

  Constructor: Constructors are not inherited by subclasses. Each class must define its own constructors.

  Method: Methods can be inherited by subclasses and can be overridden to provide specific behavior in the subclass.

- **Object Initialization:**

  Constructor: A constructor initializes an object that does not yet exist. It sets up the initial state of the object.

  Method: A method performs operations on an already existing object. It can manipulate the object's state or perform actions using the object's data.

Here is an example demonstrating the differences between a constructor and a method:

```
public class Example {
    private String name;
    private int age;
    // Constructor
    public Example(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Method
    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
    public static void main(String[] args) {
        // Creating an object using the constructor
        Example example = new Example("Jimmy", 30);
        // Calling a method on the created object
        example.displayInfo();
    }
}
```

43. What Is a Destructor?

A destructor in Object-Oriented Programming is a method that is triggered automatically right before an object's memory is deallocated. This can happen when the object’s scope ends, is contained within another object whose lifetime ends, or is dynamically allocated and explicitly freed.

OOPs Interview Questions for Experienced

Studying the Scrum Master interview questions outlined in this tutorial will equip you with the right approach to effectively prepare for this job role.

1. How to Implement Abstraction in Java?

There are two methods to implement abstraction in Java:

- Abstract class
- Interface

2. What Is an Abstract Class?

An abstract class is a template definition of methods and variables for a particular class or category of objects. In programming, objects are code units, and each object belongs to a generic class.

Abstract classes contain one or more abstract methods or behaviors. Abstracting objects or classes means summarizing their characteristics relevant to the current program's operation. Abstract classes are utilized in all Object-Oriented Programming languages, including Java, C++, C#, and VB.NET.

3. Explain the Interface.

In Object-Oriented Programming, an interface is a contract that defines a set of abstract methods and constants that a class must implement. It serves as a blueprint for classes, specifying what methods should be available without dictating how these methods should be implemented.

Characteristics of interface include:

- Interfaces contain method signatures without implementations. Classes that implement an interface must provide concrete implementations for all its methods.
- Unlike classes in many OOP languages, a single class can implement multiple interfaces, allowing for a form of multiple inheritance.
- Interfaces enable loose coupling between different parts of a system, as classes can depend on interfaces rather than concrete implementations.
- Objects of different classes implementing the same interface can be treated uniformly, enhancing code flexibility.
- Interfaces act as contracts between different system components, clearly defining expected behavior.

4. How Is an Abstract Class Different From an Interface?

This is one of the frequently encountered OOPs interview questions. The reason is that abstract classes can contain non-final variables, while variables in an interface are final, public, and static.

```
interface SimpleInterface {
int CONSTANT = 100; // implicitly public, static, and final
}
```

```
abstract class SimpleAbstractClass {
 int variable = 100; // can be non-final and instance-specific
}
```

The above example shows that interfaces can only have constants, while abstract classes can have mutable instance variables.

The following table demonstrates the difference between an abstract class and interface.

Characteristic	Abstract Class	Interface
Variables	Can have non-final instance variables.	Only constants (public static final).
Methods	Can have both abstract and concrete methods.	All methods are implicitly abstract (before Java 8).
Constructor	Can have constructors.	Cannot have constructors.
Multiple Inheritance	A class can extend only one abstract class.	A class can implement multiple interfaces.
Access Modifiers	Can use any access modifier for members.	All members are implicitly public.
Instantiation	Cannot be instantiated.	Cannot be instantiated.
Purpose	For objects that are closely related.	For unrelated classes to implement common behavior.
Speed	Slightly faster	Slightly slower due to extra indirection.
Default Method Implementation	Supported	Supported (from Java 8 onwards)
Static Methods	Can have static methods	Can have static methods (from Java 8 onwards)
Private Methods	Can have private methods	Can have private methods (from Java 9 onwards).

5. Can You Declare an Interface Method Static?

No, you cannot declare an interface method as static in most Object-Oriented Programming languages, including Java and C#. Interface methods are, by default, abstract and public. Static methods belong to the class itself, not to any specific instance. The purpose of an interface is to define a contract that implementing classes must follow, which involves instance methods.

- Interface methods are, by default, abstract and public.
- Static methods belong to the class itself, not to any specific instance.
- The purpose of an interface is to define a contract that implementing classes must follow, which involves instance methods.

However, there are some exceptions:

- In Java 8 and later, you can have static methods in interfaces, but these are not considered part of the interface contract. They are utility methods associated with the interface.
- In C# 8.0 and later, interfaces can have static members, including static methods, but these are not inherited by implementing classes.

6. Can We Make Constructors Static?

No, constructors cannot be made static in most Object-Oriented Programming languages, including Java and C#.

Constructors are specifically designed to initialize new object instances when they are created. They set up the initial state of an object and are called automatically when an object is instantiated using the new keyword.

Static members, on the other hand, belong to the class itself rather than to any specific instance of the class. They can be accessed without creating an object of the class.

Making a constructor static would contradict its fundamental purpose of initializing individual object instances. It would also conflict with object creation and initialization

in Object-Oriented Programming.

If you need class-level initialization, most languages provide other mechanisms like static initializer blocks or static methods that can be used for this purpose.

7. What Are the Rules for Creating a Constructor?

This is one of the frequently encountered OOPs interview questions. These are the rules to follow while creating constructors:

- The name of the constructor must be the same as the class name.
- Constructors in Java do not have a return type, even void.
- Multiple constructors can exist in the same class, known as constructor overloading.
- Access modifiers can be used with constructors to change their visibility/accessibility.
- A default constructor is provided by Java and invoked during object creation unless you define any constructor.

8. Can We Make the Abstract Methods Static in Java?

No, you cannot declare abstract methods as static in Java due to the inherent contradictions between the concepts of abstract and static methods.

9. Is It Possible to Declare an Abstract Method With the Private Modifier?

No, an abstract method cannot be private because it must be implemented in the child class.

Declaring it as private would prevent it from being accessed and implemented outside the class.

10. What Is the Virtual Function?

In Object-Oriented Programming, such as C++, a virtual function or method is an inheritable and overridable function dispatched dynamically. Virtual functions play a crucial role in runtime polymorphism in OOP, enabling the execution of target functions that are not precisely identified at compile time.

Many programming languages, including JavaScript, PHP, and Python, treat all methods as virtual by default and do not offer a modifier to alter this behavior. However, specific languages provide modifiers like final and private in Java and PHP to prevent methods from being overridden by derived classes.

11. What Are the Characteristics of an Abstract Class?

Abstract classes in Object-Oriented Programming have several key characteristics:

- **Incomplete Implementation:** Abstract classes can contain both fully implemented methods and abstract methods (methods without a body).
- **Cannot be Instantiated:** You cannot create objects directly from an abstract class.
- **Meant for Inheritance:** Abstract classes are designed to be subclassed by other classes that provide implementations for the abstract methods.
- **Can Have Constructors:** Although they can't be instantiated directly, abstract classes can have constructors to initialize fields.
- **May Contain Instance Variables:** Abstract classes can define and initialize instance variables.
- **Can Have any Access Modifier:** Methods in an abstract class can be public, protected, or private.
- **Support for Multiple Inheritance:** In languages like Java, abstract classes can extend another class and implement interfaces.
- **Can Have Static Members:** Abstract classes can contain static methods and variables.
- **May Contain Final Methods:** These cannot be overridden by subclasses.

- **Provide a Common Interface:** They often define a common structure for a group of related subclasses.

12. What Are the Differences Between a Copy Constructor and an Assignment Operator?

Here are the differences between a copy constructor and an assignment operator:

Aspect	Copy Constructor	Assignment Operator
Purpose	Creates a new object as a copy of an existing one.	Copies contents of one object to an existing object.
When it's called	When a new object is created.	When an existing object is assigned a new value.
Syntax (C++)	`ClassName(const ClassName& other)`	`ClassName& operator= (const ClassName& other)`
Returns	Nothing (constructs a new object).	Reference to the assigned object (usually).
Self-assignment check	Not needed	Often needed to handle a = a case.
Memory allocation	Always allocates new memory.	May or may not allocate new memory.
Usage with new	Can be used with new.	Cannot be used with new.
Inheritance behavior	Compiler-generated version is called the base class version.	Compiler-generated version doesn't call the base version.
Default implementation	Performs shallow copy.	Performs shallow copy.

13. What Is the Difference Between Abstraction and Encapsulation?

Abstraction and encapsulation are important concepts in programming. They're related but serve different purposes:

- Encapsulation encapsulates the internal workings of an object, maintaining its state and operations within a defined scope. Abstraction focuses on defining a clear and concise interface for interaction.
- Abstraction simplifies software design by focusing on essential features, promoting clarity, and reducing complexity.Encapsulation ensures that data integrity and security are maintained by controlling access to internal states and methods.
- Encapsulation manages how data is stored and accessed internally, while Abstraction simplifies interaction with complex systems by providing a clear, high-level view.
- While encapsulation protects data by encapsulating it within a class and providing controlled access methods, Abstraction simplifies the representation of systems by focusing on essential behaviors and ignoring unnecessary details.

14. How Much Memory Does a Class Occupy?

The memory occupied by a class depends on several factors:

- The size and number of data members in the class.
- Any padding added for memory alignment.
- Overhead for the class itself (e.g., vtable pointer for classes with virtual functions).

The actual memory usage can be larger than the sum of the sizes of data members due to:

- Padding between data members for alignment.
- Virtual function table pointers (in C++ for classes with virtual functions).
- Object headers (in Java).

15. What Is Coupling in OOP, and Why Is It Helpful?

Coupling in Object-Oriented Programming refers to the degree of interdependence between classes or modules in a software system. It measures how closely connected or dependent different components are on each other.

There are two types of coupling:

- Tight coupling
- Loose coupling

Difference between tight coupling and loose coupling:

- Tight coupling makes testing difficult, while loose coupling makes it easier.
- Unlike tight coupling, loose coupling promotes programming to interfaces, not implementations.
- Tight coupling makes it hard to swap code between classes; loose coupling allows for easier swapping of modules.
- Tight coupling is inflexible to changes, whereas loose coupling supports easier modifications.

16. What Is Cohesion in OOP?

Cohesion in OOP refers to the degree to which the elements within a module or class belong together and work towards a single, well-defined purpose. It measures how closely related and focused the responsibilities of a single module or class are.

17. Name the Operators That Cannot Be Overloaded.

The operators that cannot be overloaded in C++ are:

- Scope resolution operator (::)
- Ternary or conditional operator (?:)
- Member access or dot operator (.)
- Pointer-to-member operator (.*)

- Object size operator (sizeof)
- Object type operator (typeid)
- static_cast (casting operator)
- const_cast (casting operator)
- reinterpret_cast (casting operator)
- dynamic_cast (casting operator)

18. What Are the Manipulators in OOP, and How Do They Work?

Manipulators in Object-Oriented Programming, particularly in C++, are special functions or objects that modify input/output streams. They work by altering the formatting or behavior of streams without changing the actual data.

Manipulators work by modifying the state or behavior of input/output streams:

- **Function-Based Manipulators:**
  - These are functions that take a stream reference as an argument and return the same stream reference.
  - They modify the stream's internal state or flags.
  - The stream's operator *<< or operator >>* is overloaded to handle these functions.
  - When encountered in an I/O statement, the manipulator function modifies the stream.
- **Class-Based Manipulators:**
  - These are objects of classes with overloaded stream insertion/extraction operators.
  - The overloaded operators modify the stream's state.
- **Stream State Modification:**
  - Manipulators typically change internal flags or data members of the stream object.
  - These modifications affect how subsequent I/O operations behave.
- **Chaining:**

- Manipulators can be chained in a single statement because they return the stream object.
- This allows for multiple modifications in one line of code.

- **Persistence:**

- Some manipulators' effects persist until explicitly changed.
- Others (like the *setw()* function) affect only the next I/O operation.

- **Implementation in *iostream* Library:**

- Many manipulators are implemented as inline functions for efficiency.
- They often use stream member functions to modify the stream's state.

19. Give a Real-World Example of Polymorphism.

A real-world example of polymorphism can be found in a digital media player application. Let's consider a music player that can handle various types of audio files:

- **Base Class:** AudioFile

- Properties: title, artist, duration
- Methods: *play()*, *pause()*, s*top()*

- **Derived Classes:**

- **MP3File:** MP3 Audio File
- **WAVFile:** WAV Audio File
- **AADFile:** Apple Audio File

Each derived class inherits from AudioFile but implements the play() method differently due to the specific encoding and decoding requirements of each file format.

In the application:

```
//cpp
vector<AudioFile*> playlist;
```

```
playlist.push_back(new MP3File("Song1.mp3"));
playlist.push_back(new WAVFile("Song2.wav"));
playlist.push_back(new AADFile("Song3.aad"));

for (AudioFile* song : playlist) {
    song->play();  // Polymorphic call
}
```

Here, the play() method is called on each song in the playlist. The actual implementation called depends on the specific file type, demonstrating polymorphism:

- MP3File might use an MP3 decoder.
- WAVFile might directly stream the uncompressed audio.
- AADFile might use Apple's proprietary decoder.

This example shows how polymorphism allows the application to treat different audio file types uniformly through a common interface (AudioFile) while executing the appropriate play() method for each specific file type.

20. What Is the Difference Between a Base Class and a Superclass?

Aspect	Base Class	Superclass
Definition	The class from which other classes inherit.	The class from which other classes inherit.
Usage	Commonly used in C++ terminology.	Commonly used in Java terminology.
Synonyms	Parent class, superclass.	Parent class, base class.
Hierarchy	At the top or intermediate level of inheritance.	At the top or intermediate level of inheritance
Purpose	Provides common attributes and methods.	Provides common attributes and methods.
Inheritance	Other classes derive from it.	Other classes extend from it.

21. What Is Data Abstraction?

Data abstraction, the most basic form of abstraction in Object-Oriented Programming, involves manipulating complex objects where the underlying data structure or characteristics remain hidden.

22. What Are the Levels of Data Abstraction?

There are three main levels of data abstraction in OOP:

- **Physical or Implementation Level:**
  - This is the lowest level of abstraction.
  - It deals with how data is actually stored and managed in memory.
  - This level is typically hidden from the user and handled by the programming language or runtime environment.
- **Logical or Class Level:**
  - This is the intermediate level of abstraction.
  - It defines the structure and behavior of objects through classes.
  - It includes the attributes (data members) and methods (member functions) that make up a class.
  - This level is where the programmer defines the blueprint for objects.
- **View or Interface Level:**
  - This is the highest level of abstraction.
  - It defines how users or other parts of the program interact with objects.
  - This level typically includes public methods and properties that are accessible to users of the class.
  - It hides the internal implementation details and exposes only what's necessary for using the object.

23. What Are the Types of Variables in OOP?

There are three main types of variables in Object-Oriented Programming:

- Instance variables (also called object variables)
- Class variables (also called static variables)
- Local variables

24. Is It Possible to Overload a Constructor?

Yes, it's possible to overload a constructor.

25. Can We Overload the Main() Method in Java and Give an Example?

In Java, you can't truly overload the *main()* method in the sense of having multiple *main()* methods that the JVM will recognize as entry points for your program. However, you can have multiple methods named *main()* with different parameter lists within the same class. The JVM will only use the standard *main(String[] args)* as the entry point.

Here's an example to understand this:

```
public class MainOverloadExample {
    // The actual entry point recognized by JVM
    public static void main(String[] args) {
        System.out.println("Main method called with String[] args");

        // Calling other "main" methods
        main();
        main(5);
        main("Hello");
    }

    // Overloaded main methods (not entry points)
    public static void main() {
        System.out.println("Main method with no arguments");
    }

    public static void main(int number) {
        System.out.println("Main method with int argument: " + number);
```

```
    }

    public static void main(String str) {
        System.out.println("Main method with String argument: " + str);
    }
}
```

Important points to note:

- Only *public static void main(String[] args)* serves as the program's entry point.
- Other *main* methods are treated as regular methods and can be called from within the program.
- These additional *main()* methods don't affect how the program is started by the JVM.
- This approach can help organize code or provide different ways to process input, but it's not true method overloading from the JVM's perspective.

26. What Do You Understand by Copy Constructor in Java?

In Java, there is no explicit copy constructor like in C++. Instead, Java provides a default implementation of object copying through the assignment operator (=) and the constructor that takes an object of the same class as a parameter.

When you create a new object by assigning an existing object to it, Java performs a shallow copy by default. A shallow copy creates a new object and copies the reference values of the instance variables from the original object to the new object. This means that both objects will share the same reference to mutable objects (objects that can be modified after creation, such as arrays or other custom objects).

Here's an example to illustrate object copying in Java:

To create a copy constructor, start by defining a constructor that accepts an object of the same type as its parameter:

```
public class WebPage {
    private String url;
    private int loadTime;

    public WebPage(WebPage webPage) {
    }
}
```

Afterward, each field of the input object is copied into the new instance:

```
public class WebPage {
    private String url;
    private int loadTime;

    public WebPage(WebPage webPage) {
        this.url = webPage.url;
        this.loadTime = webPage.loadTime;
    }
}
```

What we have here is a shallow copy, which is suitable because all our fields—such as *String* and *int*—are either primitive types or immutable types.

If the Java class contains mutable fields, an alternative approach in its copy constructor is to perform a deep copy. This ensures that the newly created object is independent of the original one by creating distinct copies of each mutable object:

```
public class WebPage {
    private String url;
    private int loadTime;
    private Map<String, String> pageElements;

    public WebPage(WebPage webPage) {
        this.url = webPage.url;
        this.loadTime = webPage.loadTime;
        this.pageElements = new HashMap<>(webPage.pageElements);
    }
}
```

27. What Are the Differences Between a Copy Constructor and a Clone?

In Java, the clone method allows us to create an object from an existing object, but the copy constructor offers several advantages:

- Implementing a copy constructor is simpler; there's no need to handle *CloneNotSupportedException* or implement the *Cloneable* interface.
- Unlike clone, which returns a general object reference requiring typecasting, the copy constructor directly returns the appropriate type.
- Copy constructors can assign values to final fields, which is impossible with the clone method.

28. What Is the Static Variable?

When a variable is declared static, a single class-level copy is created and shared among all objects. This makes static variables effectively global variables shared across all class instances.

Some important points about static variables are:

- All objects of the class access the same copy of a static variable.
- Static variables conserve memory as they are not duplicated for each instance.
- They belong to the class rather than any specific instance.
- Static variables can be accessed without creating an object of the class.
- Static variables are declared using the *static* keyword at the class level.
- Static variables and static blocks are initialized in the order they appear in the code.
- You can access static variables directly through the class name without creating an instance of the class.

29. What Is the Static Method?

When the *static* keyword is used, methods can be created that do not require any class instances to exist. These are known as static methods.

Syntax:

```
[Access_modifier] static [return_type] methodName([parameters]) {
// Method body
}
```

30. What Are the Restrictions Applied to the Java Static Methods?

Here are some of the restrictions on Java static methods:

- **Non-Static Access:**
  - Static methods cannot directly access non-static (instance) members (methods or variables).
  - This restriction exists because static methods operate at the class level, not on specific instances.
- **'this' and 'super' Keywords:**
  - The *this* keyword refers to the current instance, which doesn't exist in a static context.
  - *super* refers to the parent class instance, which doesn't apply in a static context.
- **Data Access Limitation:**
  - Static methods can only directly access static data (static variables or other static methods).
  - A static method would need an object reference passed as a parameter to access instance data.
- **No True Overriding:**
  - Static methods cannot be overridden in subclasses.
  - A subclass can define a method with the same signature, but this is method hiding, not overriding.

31. Why Is the main() Method Static?

It is not possible to invoke a function without constructing an instance of its class, as there is no object of a class when the JVM starts. The *main()* function is declared static so that the class can be loaded by JVM into the main memory. This design allows the Java runtime to begin execution without instantiating any objects.

- Every Java program begins with the main() function. The *main()* method is essential because the compiler begins executing a program from the *main()* function only.
- The JVM should instantiate the class if the main() function is non-static.
- JVM can simply invoke static methods without generating an instance of the class by utilizing only the class name.
- The *main()* function must be declared static, and public, and must have a void return type. The JVM will be unable to run the Java program and will throw an error if we do not declare it as static and public or if we do not provide it with a void return type.

32. Can We Override Static Methods?

No, *static* methods cannot be overridden because method overriding is implemented through dynamic binding during runtime, whereas static methods are bound through static binding during compile time. Thus, we cannot override static methods. Often asked, this OOPs interview question is common.

33. What Is the Static Block?

When we use the static keyword with a block of code, it is referred to as a static block. Unlike C++, Java supports a special static block (also known as a static clause) for the static initialization of a class. This code is executed only once when the class is first loaded into memory.

Here's an example:

```
public class MyClass {
 private static int counter;
 private static final double PI;

 static {
```

```
    counter = 0;
    PI = calculatePi(); // Assume calculatePi() is a method that computes the val
    System.out.println("Static block executed");
    }

    // Rest of the class...
}
```

In this example, the *Static block* initializes the counter variable to 0 and the PI constant with the value computed by the *calculatePi()* method. The message Static block executed will be printed when the *MyClass* is loaded into memory.

34. Can a Program Execute Without the Main() Method?

Yes, a Java program can be executed without a *main()* method by using a static block.

35. What if the Static Modifier Is Removed From the Signature of the Main Method?

The removal of the static modifier from the *main()* method would result in the following problems:

- The program will fail to compile.
- Even if compilation was somehow successful, the Java Virtual Machine could not execute the program. The JVM looks explicitly for a static *main()* method as the entry point for any Java application.
- Without the static modifier, the main would become an instance method. This creates a paradoxical situation where the JVM would need to create an object of the class to call the *main()* method, but it can't create an object without already having a running program.
- The Java language specification explicitly requires the *main()* method to be static. A non-static *main()* method does not conform to these specifications, making the program invalid as a Java application.

36. What Is the Difference Between Static Methods and Instance Methods?

This is one of the most commonly asked OOPs interview questions. The following are the differences between static and instance methods:

- Instance methods have direct access to other instance methods and instance variables.
- Instance methods also have direct access to static variables and static methods.
- Static methods can directly access static variables and static methods.
- Static methods cannot directly access instance methods and instance variables; they must use an object reference. Furthermore, static methods cannot use this keyword since no instance exists to reference.

37. What Is this Keyword in Java?

The *this* keyword in Java is a reference variable that points to the current object within instance methods or constructors. It's used to distinguish between instance variables and parameters, call methods or constructors of the current class, and pass the current object as an argument.

38. What Are the Main Uses of this Keyword?

The *this* keyword has several important uses:

- When instance variables are shadowed by method or constructor parameters, this helps distinguish them.
- It can be used to call methods of the current class.
- It can be used for constructor chaining within the same class.
- It can be passed as an argument to methods.
- It can be used to return the current class instance from a method.

39. Can We Assign the Reference to this Variable?

In Java, the *this* keyword is a reference variable that refers to the current object instance. However, you cannot assign a new value to *this* keyword. This restriction is in place because *this* keyword is a final reference that always points to the current object, and allowing it to be reassigned could lead to significant confusion and errors in the code.

40. Can We Use this Keyword to Refer to Static Members?

In Java, you cannot use *this* keyword to access static members. The *this* keyword refers to the current object instance, whereas static members are class-level entities and not tied to any particular instance. Therefore, using *this* in a static context is not allowed.

41. How Can Constructor Chaining Be Done Using this Keyword?

Constructor chaining using *this* keyword is a technique where one constructor calls another in the same class. Here's how it works:

- Use *this()* call to invoke another constructor.
- The *this()* call must be the first statement in the constructor.
- Only one *this()* call can be used per constructor.

Let's illustrate this with an example:

```java
public class Car {
    private String brand, model, fuelType;
    private int year, range;

    public Car(String brand) {
        this(brand, "Unknown", 2024);
    }

    public Car(String brand, String model) {
        this(brand, model, 2024);
    }

    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;

        switch (brand.toLowerCase()) {
            case "tesla":
                this.fuelType = "Electric";
                this.range = 300;
                break;
            case "mercedes-benz":
                this.fuelType = "Gasoline";
                this.range = 400;
                break;
            case "chevrolet":
                this.fuelType = "Hybrid";
                this.range = 350;
                break;
            default:
                this.fuelType = "Unknown";
                this.range = 0;
        }
    }

    @Override
    public String toString() {
        return String.format("%s %s (%d): %s, Range: %d miles", brand, model,
year, fuelType, range);
    }

    public static void main(String[] args) {
        System.out.println(new Car("Tesla"));
        System.out.println(new Car("Mercedes-Benz", "S-Class"));
        System.out.println(new Car("Chevrolet", "Volt", 2024));
    }
}
```

This example shows how constructor chaining is done using this keyword:

- The single-parameter constructor calls the two-parameter constructor.

- The two-parameter constructor calls the three-parameter constructor.
- The three-parameter constructor initializes all fields and sets brand-specific details.

42. Which Class Is the Superclass for All the Classes?

This is a standard OOPs interview question. The superclass for all classes in Java is the Object class.

43. Why Is Multiple Inheritance Not Supported in Java?

Java doesn't support multiple inheritance of classes to avoid the "diamond problem" or "deadly diamond of death." This problem happens when a class inherits from two classes that have a common ancestor, leading to potential ambiguity and complexity.

In this scenario, if B and C both override a method from A, which version should D inherit?

This ambiguity can lead to:

- Confusion in method resolution.
- Potential runtime errors.
- Increased complexity in language design and implementation.
- Difficulties in maintaining and understanding the code.

44. What Is Super in Java?

The *super* keyword in Java is a reference variable that points to the immediate parent class object.

45. How Can Constructor Chaining Be Done by Using the Super Keyword?

Constructor chaining using the super keyword can be done to call a constructor in the superclass from a subclass.

Here's how it works:

- Use the *super()* call to invoke a constructor in the superclass.
- The *super()* call must be the first statement in the subclass constructor.
- If not explicitly called, the compiler automatically inserts *super()* to call the no-argument constructor of the superclass.

Let's understand this with an example:

```
// Parent class
class Vehicle {
    protected String brand;
    protected String model;

    public Vehicle() {
        this("Unknown", "Unknown");
    }

    public Vehicle(String brand) {
        this(brand, "Unknown");
    }

    public Vehicle(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

    public void displayInfo() {
```

```
        System.out.println("Brand: " + brand + ", Model: " + model);
    }
}

// Child class
class Car extends Vehicle {
    private int numDoors;

    public Car() {
        super();  // Calls Vehicle()
        this.numDoors = 4;
    }

    public Car(String brand) {
        super(brand);  // Calls Vehicle(String)
        this.numDoors = 4;
    }

    public Car(String brand, String model) {
        super(brand, model);  // Calls Vehicle(String, String)
        this.numDoors = 4;
    }

    public Car(String brand, String model, int numDoors) {
        super(brand, model);  // Calls Vehicle(String, String)
        this.numDoors = numDoors;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();  // Calls Vehicle's displayInfo()
        System.out.println("Number of doors: " + numDoors);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Toyota");
        Car car3 = new Car("Honda", "Civic");
        Car car4 = new Car("Ford", "Mustang", 2);

        car1.displayInfo();
```

```
        car2.displayInfo();
        car3.displayInfo();
        car4.displayInfo();
    }
}
```

In this example:

- The *Vehicle* class has three constructors demonstrating constructor chaining within the same class using *this()*.
- The *Car* class extends *Vehicle* and has four constructors:
  - The first calls *super()* to invoke the no-argument constructor of *Vehicle*.
  - The second calls *super(brand)* to invoke the one-argument constructor of *Vehicle*.
  - The third and fourth call *super(brand, model)* to invoke the two-argument constructor of *Vehicle*.
- Each *Car* constructor then initializes its own numDoors field.
- The *displayInfo()* method in Car overrides the one in *Vehicle*, calling *super.displayInfo()* to reuse the parent's implementation before adding its information.

46. What Are the Main Uses of the Super Keyword?

The *super* keyword is used for:

- Accessing superclass members
- Invoking superclass constructors
- Invoking superclass methods

47. What Are the Differences Between this and the Super Keyword?

Following are the differences between *this* and *super* keywords:

Aspect	this Keyword	super Keyword
Purpose	Refers to the current instance of the class.	Refers to the immediate parent class instance.
Usage in Constructors	Calls another constructor in the same class.	Calls a constructor in the immediate parent class.
Accessing Members	Accesses members of the current class.	Accesses members of the parent class.
Method Invocation	Invokes current class methods.	Invokes overridden methods of the parent class.
Syntax for Constructor Call	*this()* or this*(parameters)*	*super()* or *super(parameters)*
Position in Constructor	Must be the first statement if used.	Must be the first statement if used.
Applicable to	Instance methods and constructors.	Instance methods, constructors, and instance variables.
Use with Static Members	Cannot be used with static members.	Cannot be used with static members.
Implicit Use	Compiler adds this implicitly when accessing instance members.	Compiler doesn't add super implicitly (except default for the constructor call).
Overriding	Not applicable	Used to call the overridden method of the parent class
Scope	Limited to the current class.	Extends to the immediate parent class.

48. Can You Use this() and super() in a Constructor?

You cannot use *this()* and *super()* together in the same constructor.

49. What Is Object Cloning?

Object cloning is the process of creating an exact copy of an object. The *clone()* method of object class is used to clone an object. The *clone()* method provided by the object class in Java is utilized for this cloning process.

50. Why Is Method Overloading Not Possible by Changing the Return Type in Java?

This is one of the most commonly asked OOPs interview questions. Method overloading is not possible by changing only the return type in Java for the following reasons:

- If overloading were allowed based on return type, the compiler wouldn't know which method to call in situations where the return value is not used.
- Java uses type erasure for generics, which can lead to conflicts if methods differ only by return type.
- Method overloading is resolved at compile-time based on the method signature, which doesn't include the return type.
- In Java, a method's signature consists of its name and parameter list, not the return type.
- Allowing overloading by return type would break existing code and Java's commitment to backward compatibility.

51. What Is Method Overloading With Type Promotion?

When resolving overloaded methods in Java, if an exact match isn't found, the compiler first promotes lower data type arguments to higher ones. It checks for a matching method after each promotion. If a match is found, that method is used.

If not, the compiler continues promoting until all possibilities are exhausted. If no match is found after all promotions, a compile-time error occurs. This process is known as automatic type promotion in method overloading.

The promotion follows a hierarchy:

byte → *short* → *int* → *long* → *float* → *double*

char → *int*

Let's take a practical example of programs based on the automatic type promotion concept:

```
public class TypePromotionExample {
 public void display(int a) {
 System.out.println("int: " + a);
 }

 public void display(double a) {
 System.out.println("double: " + a);
 }

 public static void main(String[] args) {
 TypePromotionExample obj = new TypePromotionExample();

 byte b = 25;
 obj.display(b); // Promotes byte to int

 short s = 30;
 obj.display(s); // Promotes short to int

 char c = 'A';
 obj.display(c); // Promotes char to int

 float f = 3.14f;
 obj.display(f); // Promotes float to double
 }
}
```

In this example:

- The *byte* and *short* arguments are promoted to *int*.
- The *char* argument is promoted to *int*.
- The *float* argument is promoted to *double*.

52. Can We Change the Scope of the Overridden Method in the Subclass?

Yes, we can adjust the scope of the overridden method in the subclass. It's important to note that while increasing its accessibility, we cannot decrease it.

53. What Is the instanceOf Operator?

The *instanceof* operator is a binary operator that checks whether an object belongs to a specific type. It returns either true or false, making it a type comparison operator as it matches the instance against the type. The basic syntax of the instanceof operator is:

The basic syntax of the *instanceof* operator is:

```
(object) instanceof (type)
```

54. What Are the Advantages of Encapsulation in Java?

Encapsulation in Java, a fundamental concept in real-time programming, provides several advantages:

- Classes gain complete control over data members and methods This control ensures data integrity and prevents unintended modifications.
- It simplifies user interaction by concealing complex code implementations This reduces potential errors and improves user experience.
- It allows class variables to be set as read-only or write-only Tailored to specific needs, this flexibility protects sensitive data while enabling necessary access.
- Encapsulation facilitates the reuse of existing code Saving time and resources when developing new applications.
- Updating existing code is easier and less error-prone Crucial for maintaining and scaling complex software systems.

- Encapsulated code makes unit testing straightforward Leading to more reliable and maintainable software.
- Popular IDEs support getters and setters Enhancing coding efficiency and developer productivity.

55. How Many Types of Exceptions Can Occur in a Java Program?

In Java, exceptions are categorized into three main types:

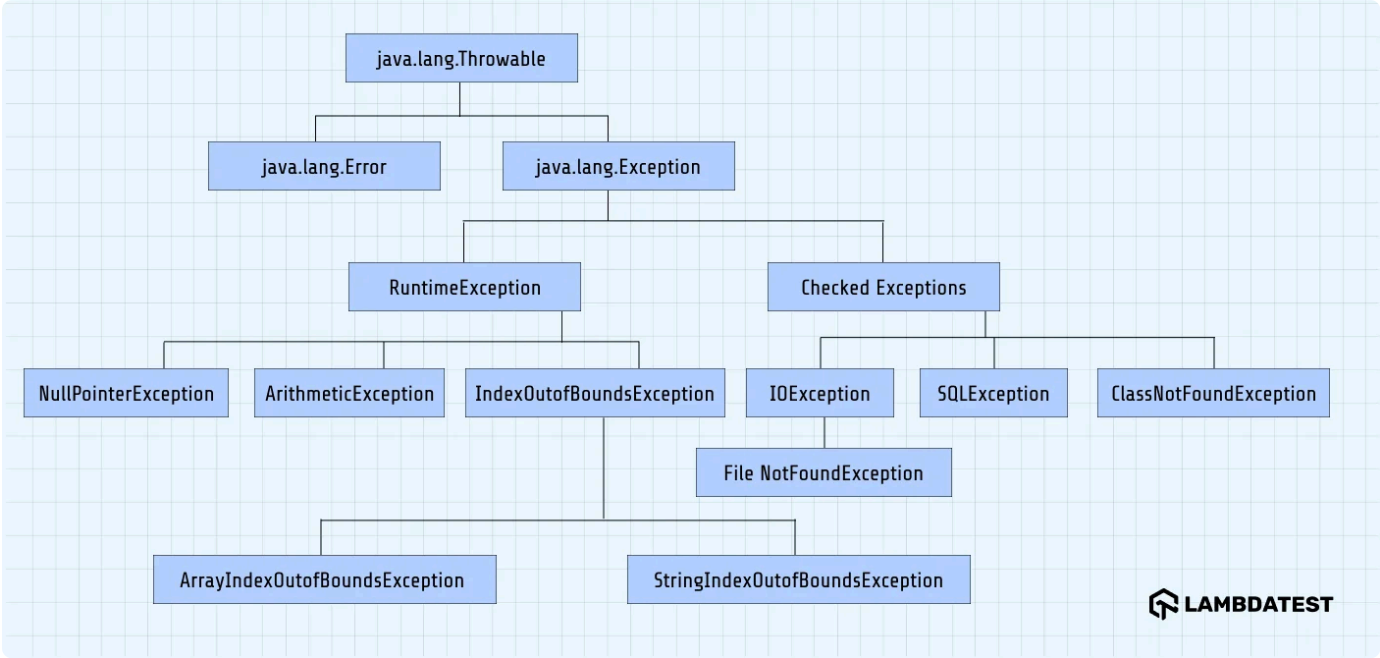
- Checked exceptions
- Unchecked exceptions (runtime exceptions)
- Errors

56. What Is Exception Handling?

Exception handling is a mechanism designed to manage runtime errors, including *ClassNotFoundException*, *IOException*, *SQLException*, *RemoteException*, and others.

57. Explain the Hierarchy of Java Exception Classes.

In Java's exception hierarchy, the *Throwable* class occupies the highest position, succeeded by its subclasses *Error* and *Exception*. The following shows the hierarchy of Java Exception classes.



It's important to differentiate between *Error* and *Exception* because they represent fundamentally different categories of issues, each requiring specific handling approaches.

An exception is an unexpected event in a program that can be managed and recovered from within the program itself. In contrast, an error represents a more severe condition that typically cannot be handled within the program as it may cause damage to the system's architecture or environment.

58. What Is the Difference Between Checked and Unchecked Exceptions?

Following are the key differences between checked and unchecked exceptions:

Aspect	Checked Exceptions	Unchecked Exceptions
Compile-Time Checking	Must be caught or declared	Not required to be caught or declared
Class Hierarchy	Subclasses of exception (excluding *RuntimeException*)	Subclasses of *RuntimeException*
Error Handling	For recoverable errors	For programming errors

When to Use	Anticipated, recoverable conditions	Unexpected failures
Examples	*IOException*, SQLException	*NullPointerException*, *ArithmeticException*
Method Signatures	Must be declared in a *throws* clause.	No need to declare.
Propagation	Must be explicitly propagated.	Automatically propagate.
Performance Impact	Can have some impact.	Generally, it has less impact.
Handling Enforcement	Enforced by the compiler.	Not enforced by the compiler.

59. What Is Finally Block?

The *finally* block in programming, found commonly in languages such as Java and C#, is a segment of code that runs regardless of whether an exception is thrown or not.

Typically used with a try-catch block, the finally block ensures critical cleanup operations, like closing files or database connections, are executed.

Syntax for a *finally* block:

```
try {
    // Code that might throw an exception
} catch (ExceptionType1 e1) {
    // Handle exception of ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle exception of ExceptionType2
} finally {
    // Code that will always execute, whether an exception occurred or not
}
```

60. What Is the Difference Between Throw and Throws?

There are many differences between throw and throws keywords. A list of differences between them is given below:

Aspect	throw	throws
Purpose	Used to explicitly throw an exception.	Used to declare that a method might throw exceptions.
Usage Location	Used inside a method.	Used in the method signature.
When It's Used	At runtime, when a specific condition is met.	At compile time, to inform the caller about possible exceptions.
Number of Exceptions	Throws a single exception at a time.	Can declare multiple exceptions.
Exception Creation	Creates and throws an actual exception object.	Does not create exceptions, it just declares possibility.
Execution	Immediately transfers control to the exception handler.	Does not immediately affect program flow.
Handling Requirement	Not required to be handled or declared.	Checked exceptions must be handled or re-declared.
Applicable to	Both checked and unchecked exceptions.	Typically used with checked exceptions.
Example	throw new *IOException("File not found");*	*public void readFile()* throws *IOException, FileNotFoundException*

61. What Is Exception Propagation?

Exception propagation refers to the process of forwarding exceptions from a called method to its caller. If such an exception isn't handled by the caller, the called method and the caller terminate execution.

62. What Is Garbage Collection?

Garbage collection is a computer programming term that describes identifying and deleting objects no longer referenced by other objects.

In simple terms, garbage collection refers to discarding objects no longer needed or referenced by any other parts of the program. It's an essential aspect of how JavaScript handles memory allocation.

63. What Are the Different Types of Garbage Collectors in Java?

Java provides four types of garbage collectors that can be chosen based on requirements:

- Serial Garbage Collector
- Parallel Garbage Collector
- Concurrent Mark Sweep (CMS) Garbage Collector
- Garbage First (G1) Garbage Collector

64. What Is the Limitation of Garbage Collection?

The main disadvantage of garbage collection is that it pauses all active threads during the memory recovery phase. Garbage collection algorithms can take significant time, ranging from seconds to minutes, to execute. This extended duration makes it challenging to schedule garbage collection routines predictably.

65. How Can an Object Be Unreferenced?

There are several ways an object can become unreferenced:

- **Nulling the Reference:** This involves explicitly setting a reference variable to null.

```
Employee e = new Employee();
e = null; // The Employee object is now unreferenced
```

- **Assigning a Reference to Another:** This method reassigns an existing reference to a different object.

```
Employee e1 = new Employee();
Employee e2 = new Employee();
e1 = e2; // The first Employee object is now unreferenced
```

- **Anonymous Objects:** These are objects created without being assigned to a reference variable.

```
new Employee(); // This Employee object is immediately unreferenced
```

- **Objects Going Out of Scope:** When a method or block ends, local variables are removed from the stack.

```
void createEmployee() {
    Employee e = new Employee();
} // 'e' goes out of scope, potentially dereferencing the object
```

- **Removing From Collections:** Objects can become unreferenced when removed from collections.

```
List<Employee> employees = new ArrayList<>();
employees.add(new Employee());
employees.remove(0); // This Employee object may now be unreferenced
```

Each of these methods can result in objects becoming unreachable from the root set, making them eligible for garbage collection. The exact timing of when these objects are collected depends on the JVM's garbage collection implementation and strategy.

66. What Is the Purpose of the finalize() Method?

The *finalize()* method is defined in the object class, which can be overridden by subclasses. The garbage collector calls the *finalize()* method before an object is collected. This method can perform essential cleanup tasks before destroying the object, such as releasing resources or detaching event listeners.

67. What Is the Difference Between Final, Finally, and Finalize?

Java uses the keywords *final*, *finally*, and *finalize* in its exception handling mechanisms, each having its specific role. Below is a list of the differences between them:

Aspect	*final*	*finally*	*finalize*
Type	keyword	block	method
Purpose	To create constants, prevent inheritance/overriding.	To execute code regardless of exception occurrence.	To perform cleanup operations before the object is garbage collected.
Usage	Can be applied to variables, methods, classes.	Used with try-catch blocks.	Defined in the object class, can be overridden.
When it Executes	Compile-time (for variables), runtime (for methods/classes).	Always executes after try-catch blocks.	Called by the garbage collector before reclaiming the object's memory.
Can be Overridden	No (when applied to methods/classes).	N/A	Yes
Execution Guarantee	N/A	Guaranteed to execute (except in specific cases).	Not guaranteed to be called.
Related to	Variables, methods, classes	Exception handling	Object life cycle