**is java pure Object-Oriented Programming Language?**

Java is not a pure Object-Oriented Programming Language. A purely object-oriented language would require that everything is an object. In Java, there are a few elements that are not objects like primitive data types and Static Methods and Variables.

Java has eight primitive data types (byte, short, int, long, float, double, char, and boolean) that are not objects. These are included for performance reasons. However, Java provides wrapper classes (Integer, Double, etc.) to treat primitives as objects when necessary.

Static methods and variables belong to the class rather than instances (objects) of the class. So Java is not a pure OOP language.

Java is strongly object-oriented, supporting encapsulation, abstraction, inheritance, and polymorphism. So, Java is strongly object-oriented programming language but not pure.

**Explain oops concepts in java?**

Object-Oriented Programming (OOP) in Java is a programming paradigm that uses objects and classes to structure the software as modular, reusable, and easy to maintain.

**Class**

A class is a blueprint or template for creating objects. It defines the data (attributes) and the behavior (methods) that the objects created from the class can have.

Class fields (Attributes) are variables that hold the data for an object.

Class methods (Functions) that define the behaviour of an object.

Class constructors are special methods that are called when an object is instantiated. They initialize the object's values i.e object state.

**Object**

An object is an instance of a class.

When a class is defined, no memory is allocated until an object of that class is created.

Objects have state (attribute values) and behavior (methods).

To create an object, you use the new keyword followed by the class constructor.

**Encapsulation**

Binding (or wrapping) code and data together into a single unit are known as encapsulation.

Data (variables) and methods (functions) that operate as a single unit or class.

It restricts direct access to some of an object’s components, which can protect the object’s state from unintended or harmful modifications.

Encapsulation is achieved through the use of access modifiers.

Access specifiers (also known as access modifiers) in Java define the visibility and accessibility of classes, methods, and members.

The public access specifier allows the class, method, or variable to be accessible from any other class in any package.

The protected access specifier allows the class members to be accessible within the same package and by subclasses even those subclasses are in different packages.

When no access specifier is specified, Java uses the default access level, this means the class, method, or variable is accessible only within its own package.

The private access specifier restricts the visibility to within the same class only. It is the most restrictive access level.

Private Members are Data fields which are declared as private to prevent direct access from outside the class.

Public Methods like getters and setters are provided to access and update the value of private fields.

**Inheritance**

Inheritance is a mechanism where one class (subclass or child class) inherits the properties and behaviors (methods) of another class (superclass or parent class).

The superclass is the base class, and the subclass inherits from the superclass.

extends Keyword is used to define the inheritance relationship.

The extends keyword is used to indicate that a class is inheriting from a superclass.

Subclasses can provide specific implementations for methods that are defined in the superclass. This is known as method overriding.

The super keyword is used to refer to the superclass. It can be used to call superclass methods and constructors.

There are several types of inheritances in java like single inheritance, multilevel inheritance, hierarchical inheritance and multiple inheritance.

Single Inheritance means a class inherits from one superclass.

Multilevel Inheritance means a class inherits from a superclass which is already inherits from another superclass.

Hierarchical Inheritance means multiple classes inherit from the same superclass.

Multiple Inheritance through interfaces means a class can implement multiple interfaces. Java does not support multiple inheritance directly i.e. a class cannot inherit from multiple classes.

In inheritance concept whenever a subclass object is created, the superclass constructor is called first. This is because the subclass may depend on the initialization performed by the superclass.

Uses of inheritance are reusing existing code and enhancing or extending it without modifying the original code.

**Polymorphism**

Polymorphism allows us to perform a single action in different ways.

If one task is performed in different ways, it is known as polymorphism.

Polymorphism allows you to define one interface and have multiple implementations.

In Java Polymorphism is mainly divided into two types such as compile time polymorphism and runtime polymorphism.

Compile time polymorphism is achieved by method overloading.

Method overloading is a feature that allows a class to have more than one method with the same name, but with different parameters. The compiler determines which method to call based on the method signature.

Run time polymorphism is achieved by method overriding.

Method overriding occurs when a subclass provides a specific implementation to a method that is already defined in its superclass. The method in the subclass should have the same name, return type, and parameters as the method in the superclass.

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than compile time. It allows a superclass reference variable to refer to a subclass object and call the overridden method of the subclass. That is why Polymorphism allows objects to be treated as instances of their parent class rather than their actual class.

Polymorphism can also be achieved using interfaces. An interface in Java can be implemented by multiple classes, and a class can implement multiple interfaces. This allows for flexible code and multiple implementations of methods.

Uses of Polymorphism are code reusability, flexibility, maintainability and extensibility. The same method can be reused across different classes. New functionality can be added without altering existing code. Code is easier to maintain and understand. Classes can be extended to include new functionality without modifying existing code.

**Abstraction**

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of the object.

It reduces complexity and increases efficiency by focusing on what an object does rather than how it does it.

Abstraction in Java is achieved using abstract classes and interfaces.

An abstract class is a class that cannot be instantiated and may contain abstract methods, which are methods without a body. These abstract methods must be implemented by subclasses.

An abstract class can have both abstract and concrete methods (methods with an implementation).

An abstract class is declared using the abstract keyword.

Abstract classes can have member variables and constructors.

Abstract methods in an abstract class must be implemented by the subclasses.

An interface is a reference type in Java, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types.

Interfaces cannot contain instance fields or constructors.

An interface is declared using the interface keyword.

All methods in an interface are abstract by default (except default and static methods).

A class that implements an interface must provide implementations for all the methods declared in the interface.

By using Interfaces we can implement multiple inheritance.

Major differences between abstract class and interface are, an abstract class can have instance variables and constructors but interface cannot have instance variables and constructors. Another difference is a class can inherit from only one abstract class but a class can implement multiple interfaces.

Uses of abstraction are simplifies the code, increase reusability, improves flexibility and enhance maintainability. By hiding the complex details, abstraction makes the code easier to understand and maintain. Abstract classes and interfaces can be reused across different parts of the application. Abstraction allows for flexible code as new functionality can be added without affecting existing code. By separating the implementation details, abstraction makes it easier to manage and update code.

**Coupling**

Coupling tells how different classes or modules interconnected each other.

High (tight) coupling means classes are heavily dependent on each other.

low (loose) coupling means classes have little knowledge of each other.

Content coupling occurs when one class directly modifies or relies on the internal workings of another class. This is highly undesirable. For example, Car is tightly coupled to Engine because it creates an Engine instance and directly interacts with engine’s method.

Common coupling occurs when multiple classes share access to global data. Changes in the shared data can lead to unexpected side effects. For example, both A class and B class are coupled through the Common class's shared data.

External coupling involves classes that depend on external systems, libraries, or hardware devices. For example, the Database class is coupled to the JDBC library and the specific database connection.

To reduce Coupling, use frameworks like Spring to inject dependencies, which reduces direct dependencies.

To reduce Coupling, depend on abstractions rather than concrete implementations. So, it’s better to depend on Interfaces and Abstract Classes.

To reduce Coupling, use Observer design pattern to reduce dependencies.

To reduce Coupling, use Service Locator design Pattern to Centralize service management to reduce the number of dependencies a class has.

To reduce Coupling, use design patterns that promote loose coupling, such as Factory, Strategy, and Command patterns.

Benefits of Loose Coupling are it’s easier to make changes in one class without affecting others. Classes can be reused in different contexts without modification. Easier to extend the system with new features or components. Easier to write unit tests for loosely coupled classes. So, it’s easy to maintainability, reusability, scalability and testability.

**Cohesion**

Cohesion refers to how closely related and focused the responsibilities of a single module, class, or method.

High cohesion within a module means that the elements of the module are strongly related to each other, serving a single, well-defined purpose.

Functional cohesion occurs when parts of a module are grouped because they all contribute to a single, well-defined task. This is the highest and most desirable level of cohesion. For example, the OrderProcessor class has high cohesion because all its methods are related to processing an order method.

Sequential cohesion occurs when elements are grouped because the output from one part is the input to another part. For example, the methods in DataTransformer are sequentially cohesive because the output of process is used as input to format.

Communicational cohesion occurs when elements are grouped because they operate on the same data set. For example, the CustomerManager class has communicational cohesion because its methods operate on the same customers list.

Procedural cohesion occurs when elements are grouped because they always follow a specific sequence of execution. For example, the methods in Transaction are procedurally cohesive because they must be executed in a specific sequence.

Temporal cohesion occurs when elements are grouped because they are related by the timing of their execution. For example, the methods in InitializationManager are temporally cohesive because they are all part of the initialization process. It is almost same as procedural cohesion.

Logical cohesion occurs when elements are grouped because they logically belong to the same category, even though they may not be related functionally. For example, the methods in MathUtilities are logically cohesive because they all perform mathematical operations.

Coincidental cohesion occurs when elements are grouped arbitrarily without any meaningful relationship. This is the lowest and least desirable level of cohesion. For example, the methods in Utility have coincidental cohesion because they have no meaningful relationship to each other.

Benefits of High Cohesion are makes it easier to understand and maintain a module because all related functionalities are grouped together. Highly cohesive modules are more likely to be reusable because they serve a single purpose. Modules with high cohesion are easier to test because they have clear, well-defined responsibilities. Changes in a highly cohesive module are less likely to affect other parts of the system.

To improve Cohesion, ensure each class or method has only one reason to change. This is called Single Responsibility Principle (SRP).

To improve Cohesion, improve modular design by breaking down large classes or methods into smaller, more focused ones.

To improve Cohesion, use descriptive names that reflect the purpose of classes and methods for clear naming.

To improve Cohesion, do continuously refactor code to improve cohesion by grouping related functionalities together.

**Association**

Association represents a relationship between objects.

Association represents a relationship between two or more objects and their interaction.

In a unidirectional association, one class knows about (or uses) another class, but the reverse is not true. For example, Car class knows about Driver class, but Driver class does not know about Car class.

In a bidirectional association, both classes know about and use each other.

Aggregation is a special form of association that represents a "has-a" relationship where the contained object can exist independently of the container. For example, Department aggregates Teacher objects, but Teacher objects can exist independently of Department.

Composition is a stronger form of aggregation that represents a "has-a" relationship where the contained object cannot exist independently of the container. For example, Library composes Book objects, and Book objects cannot exist independently of Library.

**what is is-a and has-a relation in java?**

The "is-a" relationship in Java is based on inheritance. It means that an object of a subclass can be treated as an object of the superclass. This relationship is represented using inheritance by extends keyword for class and implements keyword for interface.

For example, Dog is a subclass of Animal, so we can say that a Dog is an Animal.

The "has-a" relationship in Java is based on composition or aggregation. It means that one class contains a reference of another class. We can use another class fields and methods by using their object.

For example, Car has an Engine. Here Car is a class that has an Engine class object internally and use engine fields and methods by engine object.

**Explain method overloading?**

Method overloading allows a class to have more than one method with the same name with different parameters.

The compiler determines which method to call based on the method signature at compile time.

This is a type of polymorphism in Java, specifically compile-time (or static) polymorphism.

The parameters must differ in type, number, or both.

The return type alone is not sufficient to distinguish overloaded methods.

we can overload methods by changing the number of parameters or by changing the type of parameters.

Overloading improves code readability and usability by allowing the same method name to be used for different types or numbers of arguments.

Overloading methods with similar parameter types or sequences can lead to ambiguity and errors. Ensure that the overloaded methods are clearly distinguishable.

When overloading methods, consider backward compatibility. Changing method signatures in existing code might affect other parts of the codebase.

**Explain method overriding?**

Method overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass.

This is a form of polymorphism, specifically runtime (or dynamic) polymorphism.

The overriding method must have the same name, return type, and parameters as the method in the superclass.

The access level of the overriding method cannot be more restrictive than the overridden method. For example, if the superclass method is public, the overriding method cannot be protected or private.

It is a good practice to use the @Override annotation to indicate that a method is intended to override a method in the superclass. This helps catch errors at compile time if the method does not correctly override the superclass method.

The overriding method can throw any unchecked (runtime) exceptions, regardless of whether the overridden method throws exceptions. However, if the overridden method throws checked exceptions, the overriding method can only throw the same exceptions or subclasses of those exceptions.

Only instance methods can be overridden. Static methods cannot be overridden (they can be hidden).

Overriding supports dynamic method binding (late binding), where the method to be executed is determined at runtime based on the actual object.

The super keyword is used within the subclass to refer to the superclass's implementation of an overridden method. This is useful when you want to extend or modify the behavior of the superclass method.

Use of Overriding is reuse of methods in the superclass, reducing code duplication and improving maintainability.

**what are default methods in java?**

Default methods in Java are a feature introduced in Java 8 that allows interfaces to have methods with implementations. Before Java 8, interfaces could only declare abstract methods that had to be implemented by classes implementing the interface. Default methods provide a way to add new methods to existing interfaces without breaking compatibility with classes that already implement those interfaces.

Default methods provide a default implementation in the interface itself. These methods are marked with the default keyword.

Classes that implement the interface can choose to override default methods to provide their own implementation. If not overridden, they inherit the default implementation from the interface.

Default methods can have any access modifier (public, private, protected, package-private), but they cannot be static or final.

**what are static methods in java?**

Static methods in Java belong to the class rather than to any instance of the class.

They can be called using the class name itself, without creating an instance of the class.

Static methods can access and modify static variables (class variables) that belong to the class.

Static methods are not overridden like instance methods. Instead, they are hidden when a subclass defines a static method with the same signature.

**what is constructor in java?**

a constructor is a special type of method that is automatically called when an instance (object) of a class is created using the new keyword.

Constructors initialize the newly created object by assigning initial values to instance variables and performing other initialization tasks.

Constructors are automatically invoked when an object of the class is created using the new keyword.

Constructors have the same name as the class in which they are declared.

Constructors do not have a return type, not even void.

Like methods, constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists.

**what are types of constructors in java?**

Constructors are categorized into several types based on their parameters and usage.

Default Constructor:

A constructor with no parameters is called default constructor.

If no explicit constructor is defined in a class, Java provides a default constructor automatically. It initializes the instance variables to their default values (numeric types to 0, boolean to false, object references to null).

Parameterized Constructor:

A constructor with parameters is called parameterized constructor.

Parameterized constructor allows initialization of instance variables with specific values provided during object creation.

Copy Constructor:

A constructor that initializes an object by copying another object of the same class is called copy constructor. It is useful for creating a new object with the same state as an existing object.

**Explain about private constructor?**

A constructor declared with private access modifier used to prevent instantiation of a class from outside and restricts creation to within the class itself like singleton pattern.

**Explain static block constructor?**

Static block can be used for static initialization of a class. It executed only once when the class is loaded into memory.

**what is mutable and immutable classes, methods, variables in java?**

The concepts of mutable and immutable apply to classes, methods, and variables, describing whether they can be changed after creation.

A mutable class is one whose instances can have their state (i.e., instance variables) modified after instantiation. This typically involves providing setter methods or directly modifying instance variables.

An immutable class is one whose instances cannot be modified after instantiation. Once created, the state of an immutable object cannot be changed. Typically, immutable classes set their state in the constructor and do not provide setter methods to modify instance variables.

A mutable method is one that can modify the state of an object (instance variables) on which it operates. It may alter the internal state or invoke other methods that modify the state.

An immutable method is one that does not modify the state of the object on which it operates. It typically performs computations or returns a result based on the object's state without changing it.

A mutable variable is one whose value can be changed after it is initialized. non-final variables are mutable and can be reassigned.

An immutable variable is one whose value cannot be changed once it is initialized. In Java, variables declared as final are immutable and cannot be reassigned after initialization.

**What are the advantages of immutability?**

Immutable objects are inherently thread-safe because their state cannot change once created. Immutability can lead to increased memory usage if frequent modifications require new object instances.

**what is package in java?**

A package is a mechanism for organizing classes and interfaces into namespaces. It helps to avoid naming conflicts and provides a way to logically group related classes and interfaces.

Facilitates modular programming by grouping related classes and interfaces together.

To create a package, you use the package keyword followed by the package name as the first statement in a Java source file.

To use classes from a package in another Java file, you typically import them using the import statement.

Packages follow a naming convention that uses reverse domain name notation. This convention helps ensure package names are unique and avoid conflicts.

Classes within the same package can directly access each other for public, default and protected. Classes outside the package can only access public members.

**what are non-access modifiers in java?**

Non-access modifiers provide functionality beyond the scope of access control.

They modify the behavior of classes, methods, and variables, allowing you to specify aspects such as immutability, concurrency, and inheritance.

static, final, abstract, synchronized, volatile, transient, native and strictfp are comes under non-access modifiers.

**Static**

Static non-access modifier used for memory management primarily. Memory allocation happens once when the class is loaded.

The static keyword can be applied to variables, methods, blocks, and nested classes.

When a member is declared static, it belongs to the class rather than to any particular instance of the class.

Static variables are also known as class variables. They are shared among all instances of a class. If one instance modifies the static variable, the change will be reflected across all instances.

Static members accessible using the class name only.

Static methods belong to the class rather than any particular object instance. They can be called without creating an instance of the class.

Static methods can access static variables and static methods directly.

Static methods cannot access instance variables or instance methods directly.

Static blocks are used to initialize static variables. This block gets executed when the class is loaded into memory.

Static blocks are executes only once when the class is loaded. These are used for complex static variable initialization.

Static nested classes (also known as static inner classes) are defined within another class but are not associated with instances of the outer class.

Static nested classes can access static members of the outer class.

Static nested classes cannot directly access non-static members of the outer class.

**final**

Final can be used with classes, methods, and variables to enforce constraints such as immutability and preventing inheritance or method overriding.

A class declared as final cannot be subclassed. This is used to prevent inheritance and ensure that the class's behaviour remains unchanged through inheritance. So final classes cannot be extended by other classes.

A method declared as final cannot be overridden by subclasses. This is used to prevent modification of the method's implementation by subclasses.

A variable declared as final cannot be changed once it is assigned. This makes the variable a constant.

Static Final Variable acts as a constant across all instances.

A method parameter declared as final cannot be modified within the method.

**abstract**

The abstract keyword is a non-access modifier that can be applied to classes, methods, and in some cases, variables.

An abstract class cannot be instantiated on its own will cause a compile-time error. It serves as a blueprint for other classes to inherit from. Abstract classes can contain abstract methods (methods without a body) that subclasses must implement.

An abstract method is declared without a body and is intended to be implemented by subclasses. These must be declared in an abstract class only. These should be implemented (overridden) by any non-abstract subclass.

Abstract variables do not exist directly. Variables are inherently concrete by nature and do not require abstraction. However, variables within interfaces are implicitly public, static, and final, effectively making them constants rather than abstract in the usual sense.

**Synchronized**

The synchronized keyword is a non-access modifier primarily used for multithreading to control access to critical sections of code.

It ensures that only one thread can access the synchronized method or block at a time. So, it preventing thread interference and provides thread safety.

Other threads attempting to execute the synchronized method must wait until the current thread releases the lock (intrinsic lock) on the method.

For example, the deposit and withdraw methods are synchronized to ensure that only one thread can modify the balance variable at a time. This prevents inconsistencies that could arise if multiple threads attempt to deposit or withdraw funds simultaneously.

Synchronized blocks allow finer-grained control over synchronization by locking on a specific object's monitor. This is useful when you want to synchronize only a specific portion of code rather than the entire method.

Synchronized blocks allows multiple synchronized blocks to execute concurrently as long as they lock on different objects.

The purpose of synchronized keyword is to provide thread safety, prevents race condition and ensures data visibility among multiple threads.

Careful use of synchronization is essential for optimal performance in multithreaded applications.

Improper use of synchronization can lead to deadlock, where two or more threads are blocked indefinitely, waiting for each other to release locks.

**Volatile**

The volatile keyword is a non-access modifier that is used to indicate that a variable's value may be modified by multiple threads concurrently.

When a variable is declared as volatile, any write to that variable by one thread is immediately visible to other threads.

Use volatile for boolean flags or status variables that are read and written by multiple threads.

Use volatile for simple variables (like counters) where atomicity is not required but visibility across threads is crucial.

The main purpose of volatile is to ensure visibility and ordering of variables across threads.

In a multithreaded environment, each thread may have its own copy of variables in the CPU cache. Without synchronization mechanisms like volatile, changes made by one thread to shared variables may not be immediately visible to other threads. volatile ensures that changes are visible across threads.

While volatile ensures visibility, volatile is no atomic. it does not provide atomicity for compound actions (like incrementing a variable). For atomic operations, you should use synchronized blocks or classes from the java.util.concurrent.atomic package.

Unlike synchronized, volatile does not acquire any locks. It only guarantees visibility.

**transient**

The transient keyword is a non-access modifier used in serialization to indicate that a field should not be serialized.

When a field is marked as transient, it is ignored by the serialization mechanism. During deserialization, transient fields are restored to their default values (null for objects, zero/false for primitive types).

Serialization is the process of converting an object into a stream of bytes for storage or transmission.

Certain fields of an object contain sensitive or unnecessary information that should not be persisted or transmitted. Transient is to exclude specific fields from the serialization process.

Transient provides security so exclude fields that contain sensitive information (like passwords or encryption keys) from being serialized and potentially exposed.

Transient provides optimization so exclude fields that are derived from other fields or are unnecessary for reconstructing the object state.

**native**

Methods declared as native do not provide a method body in Java. Instead, the implementation is provided by external code written in another programming language (like C or C++).

Native methods facilitate integration between Java applications and platform-specific libraries or system functions that are not directly accessible through Java's standard libraries.

It provides system level access. It provides access to operating system functions or hardware resources that are not exposed through Java's standard library.

It implements critical algorithms or operations in a lower-level language for improved performance.

**strictfp**

The main purpose of strictfp is to achieve platform-independent results for floating-point calculations.

The strictfp keyword is a non-access modifier that applies to classes, interfaces, and methods to ensure that floating-point calculations yield the same results across all platforms, regardless of the underlying hardware or operating system.

When a class or interface is declared with strictfp, all methods in that class/interface, and its subclasses/subinterfaces, are also implicitly strictfp.

When a method is declared with strictfp, all floating-point calculations within that method are performed strictly according to IEEE 754 rules, ensuring consistent results across different platforms.

Strictfp used for precise and consistent results in financial calculations involving floating-point numbers.

Strictfp used to maintaining accuracy in scientific simulations or calculations where consistent behavior across platforms is critical.

**What are wrapper classes in java?**

wrapper classes provide a way to use primitive data as objects.

Each primitive type has a corresponding wrapper class in the java.lang package.

Wrapper classes allow primitive values to be treated as objects and enabling their use in generic data structures.

Wrapper classes provide several utility methods for converting between types, parsing strings, and performing other operations.

Instances of wrapper classes are immutable, meaning their values cannot be changed once they are created.

Some wrapper classes cache frequently used to improve performance by reusing existing instances.

Java provides automatic conversion between primitive types and their corresponding wrapper classes. This feature is known as autoboxing and unboxing.

Automatic conversion of a primitive type to its corresponding wrapper class is called **autoboxing**.

Automatic conversion of a wrapper class instance to its corresponding primitive type is called **unboxing**.

**what are the ways of object creation in java?**

The most common way to create an object is by using the new keyword, which allocates memory for the object and calls the constructor to initialize it.

Using Class.forName() this method is useful when you want to create an object using the class name as a string.

The clone() method creates a new instance of a class by copying an existing instance. The class must implement the Cloneable interface and override the clone() method.

Deserialization involves creating an object from a byte stream.

Object can create by using reflection api like class.getDeclaredConstructor().newInstance().

Factory methods are static methods that return an instance of a class so we can create in this way also.

**What is object cloning in java?**

Object cloning in Java is the process of creating an exact copy of an existing object. This is done using the clone() method, which is part of the Object class.

The object that needs to be cloned must implement the Cloneable interface, and its clone() method must be overridden to enable cloning.

Cloneable interface is a marker interface that indicates the class can be cloned.

Shallow cloning creates a new instance of the object and copies all the fields. If the fields are references to objects, only the references are copied, not the actual objects. Both the original and the cloned object would reference the same instance of that object.

Deep cloning involves not only cloning the object itself but also cloning the objects referenced by the fields of the object. This ensures that the cloned object and the original object do not share references. To achieve deep cloning, you must manually clone the referenced objects.

Disadvantages for cloning are Implementing deep cloning can be complex and error-prone. If a superclass does not implement Cloneable or properly handle cloning, subclasses cannot be cloned reliably. Final fields cannot be reassigned, complicating cloning.

**Explain Autoboxing and Unboxing?**

Java provides automatic conversion between primitive types and their corresponding wrapper classes. This feature is known as autoboxing and unboxing.

Automatic conversion of a primitive type to its corresponding wrapper class is called autoboxing.

Automatic conversion of a wrapper class instance to its corresponding primitive type is called unboxing.

**Explain Shallow copy and deep copy?**

Shallow cloning creates a new instance of the object and copies all the fields. If the fields are references to objects, only the references are copied, not the actual objects. Both the original and the cloned object would reference the same instance of that object.

Deep cloning involves not only cloning the object itself but also cloning the objects referenced by the fields of the object. This ensures that the cloned object and the original object do not share references. To achieve deep cloning, you must manually clone the referenced objects.

**what is marker interface in java?**

A marker interface in Java is an interface with no methods or constants declared within it. It is used to signal or "mark" a class that implements the interface with a specific property or behaviour.

Marker interfaces act as tags to categorize or label a class, indicating that it possesses certain properties or capabilities.

They enable type checking at compile-time or runtime, ensuring that only objects of classes with the desired properties are used in certain contexts.

They can trigger specific behavior in the Java runtime or frameworks, such as enabling serialization for Serializable classes or allowing cloning for Cloneable classes.

Alternative to Marker Interfaces are Annotations. Annotations, which provide more flexibility and can carry additional metadata.

**What are existed marker interfaces in java?**

Some well-known marker interfaces in Java are Serializable, Cloneable, Remote.

Serializable indicates that a class can be serialized, which means its instances can be converted into a byte stream and subsequently reconstructed.

Cloneable indicates that a class supports cloning, meaning its instances can be copied using the clone() method.

Remote used in Java RMI (Remote Method Invocation) to indicate that a class can be used for remote communication.

**what is anonymous class and methods in java?**

In Java, anonymous classes and methods provide a way to create instances of classes and implement methods on-the-fly, without explicitly defining a class.

An anonymous class is a local class without a name. It is used to instantiate objects with certain modifications, usually as part of the code where it's defined. This allows for a quick and concise implementation of interfaces or extension of classes.

Java does not support anonymous methods in the same way it supports anonymous classes. However, Java 8 introduced lambda expressions, which provide a way to implement methods of functional interfaces (interfaces with a single abstract method) more concisely.

**what is inner class and nested class in java?**

In Java, inner classes and nested classes provide a way to logically group classes that are only used in one place, increasing encapsulation and readability.

A nested class is any class whose declaration occurs within the body of another class or interface.

A static nested class is a static member of the outer class. It can access the outer class's static members, but it cannot access the outer class's instance members or methods.

A non-static nested classes are inner classes those associated with an instance of the outer class. They can access the outer class's instance members, including private members.

Anonymous inner classes are declared and instantiated all at once using the new keyword. They are used when you need to create an instance of a class that may override some methods without having to actually create a subclass.

**what is pojo class?**

POJO(Plain Old Java Object) encapsulate their fields by providing private fields and public getter and setter methods.

POJOs are often used as data transfer objects (DTOs), entity classes in persistence frameworks, and in many other scenarios where simple data encapsulation is required.

POJOs are easy to read, write, and understand. It has simple structure makes them easy to test using standard unit testing frameworks.

POJOs have no specific framework dependency, making them versatile and reusable.

POJOs have a default constructor and a parameterized constructor.

POJOs do not require any special annotations or interfaces.

POJOs implement the Serializable interface to allow their instances to be serialized and deserialized.

**what are method parameters?**

Method parameters are the variables that are passed to a method when it is called. They allow a method to accept input values from the caller and use these values within the method to perform a specific operation.

Method parameters are defined in the method's signature, within the parentheses following the method name.

Method parameters are two types Formal Parameters, Actual Parameters (Arguments).

Formal Parameters are the parameters defined in the method declaration. They act as placeholders for the values that will be passed to the method when it is called.

Actual Parameters (Arguments) are the actual values or variables passed to the method when it is called.

Java uses pass-by-value for passing parameters to methods.

When a primitive data type (int, float, etc.) is passed to a method, a copy of the value is made. Changes to the parameter within the method do not affect the original value.

When a reference data type (objects, arrays) is passed to a method, a copy of the reference (not the object itself) is made. Changes to the object’s fields within the method will affect the original object.

Example:

// Method to modify primitive data type

public void modifyPrimitive(int num) {

num = 10; // Changes will not affect the original variable

}

// Method to modify reference data type

public void modifyObject(Person person) {

person.setName("John"); // Changes will affect the original object

}

**What are varargs(Variable Arguments)?**

Java provides a special feature called varargs (variable-length argument lists) that allows a method to accept zero or more arguments of a specified type. Varargs are specified by three dots (...) in the method parameter list.

**what is pass by value and pass by reference in java?**

Java uses pass-by-value for all method parameter passing. This means that when a method is called, the arguments are copied and these copies are passed to the method.

Pass-by-value works differs for primitive data types and reference data types.

For primitive data types (such as int, float, boolean, etc.), the actual value is copied. This means that changes made to the parameter inside the method do not affect the original value.

For reference data types (such as objects and arrays), the reference (memory address) is copied. This means that the method receives a copy of the reference, not the actual object. Changes made to the object through this reference will affect the original object because both the original reference and the copied reference point to the same object in memory.

Java does not uses pass-by-reference.

**can we overload main method?**

Yes, you can overload the main method in Java.

You can define multiple main methods with different parameter lists, but these overloaded versions are treated like any other static methods and are not used as entry points by the JVM.

The JVM calls the main(String[] args) method. Inside this method, you can call the overloaded main methods manually.

**can we override main method in java?**

No, you cannot override the main method in Java.

Since the main method is a static method, it belongs to the class rather than an instance of the class, and static methods cannot be overridden.

**can we overload static methods in java?**

Yes, you can overload static methods in Java. We can overload both instance methods and static methods.

**can we override static methods in java?**

No, you cannot override static methods in Java. In Java, static methods belong to the class itself not belongs to instances of the class.

When a static method is defined in a subclass with the same signature as one in its superclass, it hides the superclass's static method rather than overriding it. This concept is called method hiding.

**can we overload abstract methods in java?**

Yes, you can overload abstract methods in Java.

**can we override abstract methods in java?**

Yes, you can override abstract methods in Java.

**can we extend one abstract class with another abstract class in java?**

Yes, you can extend one abstract class with another abstract class in Java. This allows you to build a hierarchy of abstract classes where each class can define abstract methods that must be implemented by concrete subclasses.

**can we implement one abstract class with another abstract class?**

No. one abstract class cannot directly **implement** another abstract class. This is because Java does not support multiple inheritance for classes.

**Can we implement number of interfaces by using abstract class?**

Yes, an abstract class can implement one or more interfaces.

**can we create instance for abstract class in java?**

No, we cannot create an instance of an abstract class in Java directly using the new keyword. Abstract classes are designed to be incomplete and serve as base classes for other classes to extend and provide concrete implementations for their abstract methods.

**can we create constructors in abstract class?**

Yes, you can create constructors in an abstract class in Java. Constructors in abstract classes serve the purpose of initializing the state of the object when a concrete subclass is instantiated.

**can we create constructors in interface?**

No, you cannot create constructors in interfaces in Java. Interfaces are purely abstract and do not have instance variables or constructors like classes do.

**can we overload default methods in java?**

Yes, you can overload default methods in Java interfaces.

**can we override default methods in java?**

Yes, you can override default methods in Java interfaces.