**Java OOPs**

**─**

Contents

[**1. What is OOPs?** 2](#_Toc183285247)

[# Benefits of OOPs 2](#_Toc183285248)

[# Main features of OOPs 2](#_Toc183285249)

[**2. Basic Terminologies** 3](#_Toc183285250)

[# Class 3](#_Toc183285251)

[# Attributes and methods 3](#_Toc183285252)

[# Constructor 4](#_Toc183285253)

[# Destructor 9](#_Toc183285254)

[# Static Keyword 10](#_Toc183285255)

[**3. Encapsulation** 14](#_Toc183285256)

[# Access Modifiers 14](#_Toc183285257)

[# Getters and Setters 14](#_Toc183285258)

[**4. Inheritance** 16](#_Toc183285259)

[# Super Keyword 16](#_Toc183285260)

[# Types of Inheritance 18](#_Toc183285261)

[**4. Interview Questions** 19](#_Toc183285262)

## **1. What is OOPs?**

Object Oriented Programming is a programming paradigm that allows you to model real world entities as objects and define their behaviour through methods and attributes. Oops promote modularity, reusability and maintainability of code.

### # Benefits of OOPs

1. Modularity: - Breaks complex problem into manageable parts.
2. Reusability: - By use of inheritance.
3. Encapsulation: - Hide internal details of objects, making code more robust and secure.
4. Code readability: - Makes code easier to read.
5. Maintenance: - Easier to modify code as changes in one part of codebase have limited impact on other parts.
6. Collaboration: - Teams can work concurrently on different class or modules.
7. Scalability: - You can create new class or extend existing class without disrupting existing code.

### # Main features of OOPs

|  |  |
| --- | --- |
| **Encapsulation** | Bundling data (attribute) and methods that operate on the data into a single unit (object) while hiding internal details.  E.g. Smartphone has various components like processor, memory, camera and battery encapsulated with device’s outer shell. |
| **Abstraction** | Abstraction simplifies complex systems by showing essential details while hiding unnecessary complexities.  Abstraction is about expressing external simplicity while encapsulation is about hiding internal complexity.  E.g. For smartphones users have button and touchscreen to access different features is example of abstraction. |
| **Inheritance** | Creating new classes by inheriting attributes and methods from existing ones, promoting code reuse and hierarchy.  E.g. Car, bike, truck can extend features of Vehicle class. |
| **Polymorphism** | Refers to process by which some code, data, method or object behaves differently under different conditions.  E.g. A person can be sometimes a father and sometime a employee. |

## **2. Basic Terminologies**

Object Oriented Programming is a programming paradigm that allows you to model real world entities as objects and define their behaviour through methods and attributes. Oops promote modularity, reusability and maintainability of code.

### # Class

A class is a building block of OOP. It is a user defined data type that contains the data members and member functions that operates on the data members. It is like a blueprint or template of objects having common properties and methods.

Object refers to instance of class, which contains the instance of members and behaviour defined in the class template. Object is like the actual entity to which user interacts.

### # Attributes and methods

Attributes are variables that hold data specific to each object, while methods are functions that define behaviour of the object.

//class creation  
class Car{  
 String color;  
 int seats;  
 int maxSpeed;  
  
 public void run(){  
 System.*out*.println("Car can run on max speed of "+ this.maxSpeed);  
 }  
}

//Object creation  
public class LearnClass {  
 public static void main(String args[]) {  
 Car alto = new Car();  
 alto.color = "Black";  
 alto.seats = 4;  
 alto.maxSpeed = 80;  
 alto.run();  
 }  
}

this keyword: - The this keyword in Java is a reference to the current instance of the class. It’s used to differentiate between instance variables (also known as fields) and parameters, access methods, or even call other constructors within the same class.

* this cannot be used in static methods, as this is associated with the current instance, and static methods do not operate on instance data.
* this() (constructor chaining) must always be the first line in a constructor if used.
* Using this in nested or inner classes can access both the inner and outer class instances, depending on whether this or OuterClass.this is used.

### # Constructor

A constructor is a block of code that initializes the newly created object. In java Constructor name should be same as of class.

Java has two main types of constructors:

* Default Constructor (no-argument constructor)
* Parameterized Constructor

1. Default Constructor

If no constructor is defined, Java provides a default no-argument constructor that initializes instance variables to default values (like 0, null, or false).

class Animal {  
 String type;  
 int speed;  
  
 // Default Constructor  
 public Animal() {  
 type = "mammal";  
 speed = 10;  
 }  
  
 public static void main(String[] args) {  
 Animal myCar = new Animal(); // Calls the default constructor  
 }  
}

1. Parameterized Constructor

A parameterized constructor allows us to pass arguments to the constructor to initialize an object with specific values.

class Animal {  
 String type;  
 int speed;  
  
 // Default Constructor  
 public Animal(String type, int speed) {  
 this.type = type;  
 this.speed = speed;  
 }  
  
 public static void main(String[] args) {  
 Animal myCar = new Animal("fish",30); // Calls the default constructor  
 }  
}

Constructor Overloading

In Java, we can have multiple constructors with different parameter lists. This is called **constructor overloading**.

class Car {  
 String brand;  
 int speed;  
  
 // Default Constructor  
 public Car() {  
 brand = "Unknown";  
 speed = 0;  
 }  
  
 // Parameterized Constructor  
 public Car(String carBrand) {  
 brand = carBrand;  
 speed = 0;  
 }  
  
 // Another Parameterized Constructor  
 public Car(String carBrand, int carSpeed) {  
 brand = carBrand;  
 speed = carSpeed;  
 }  
  
 public void displayInfo() {  
 System.*out*.println("Brand: " + brand + ", Speed: " + speed);  
 }  
  
 public static void main(String[] args) {  
 Car defaultCar = new Car(); // Calls the default constructor  
 Car brandedCar = new Car("Honda"); // Calls the one-parameter constructor  
 Car customCar = new Car("BMW", 200); // Calls the two-parameter constructor  
  
 defaultCar.displayInfo();  
 brandedCar.displayInfo();  
 customCar.displayInfo();  
 }  
}

Constructor Chaining

Constructor chaining allows one constructor to call another constructor using this().

class Car {  
 String brand;  
 int speed;  
  
 public Car() {  
 this("Default Brand", 0); // Calls the parameterized constructor  
 }  
  
 public Car(String brand, int speed) {  
 this.brand = brand;  
 this.speed = speed;  
 }  
  
 public void displayInfo() {  
 System.*out*.println("Brand: " + brand + ", Speed: " + speed);  
 }  
  
 public static void main(String[] args) {  
 Car myCar = new Car(); // Calls the default constructor  
 myCar.displayInfo();  
 }  
}

Private Constructor

A constructor can also be private. Private constructors prevent object creation from outside the class and are useful for singleton patterns.

class Singleton {  
 private static Singleton *instance*;  
  
 // Private constructor  
 private Singleton() {  
 System.*out*.println("Singleton Instance Created");  
 }  
  
 public static Singleton getInstance() {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Singleton s1 = Singleton.*getInstance*();  
 Singleton s2 = Singleton.*getInstance*();  
  
 System.*out*.println(s1 == s2); // Output: true  
 }  
}

Key Points to Remember:

* Private Constructor: Prevents direct instantiation.
* Static Method: Provides a global point of access.
* Thread Safety: Ensures that only one instance is created, even in multi-threaded environments.
* Lazy Initialization: The instance is created only when it's first needed.

A Singleton design pattern ensures that a class only has one instance and provides a global point of access to it. This pattern is essential in scenarios where you need to control object creation and ensure that only one instance of a class exists.

**Why Use Singleton Pattern?**

1. **Controlled Resource Usage:**
   * **Limiting Database Connections:** A single database connection can be shared across the application to optimize resource usage.
   * **Managing Configuration Settings:** A single configuration object can be accessed globally to ensure consistent settings.
2. **Global Point of Access:**
   * **Logger:** A single logger instance can be used to log events from different parts of the application.
   * **Cache:** A single cache instance can be used to store frequently accessed data.
3. **Thread Safety:**
   * A well-implemented Singleton can be thread-safe, ensuring that multiple threads can access the same instance without causing concurrency issues.

Copy Constructor

A copy constructor is a special constructor that creates a new object by copying the values of an existing object of the same class. It is used to create a clone of an object, and the most common use case is for deep copying of objects.

class Car {  
 String brand;  
 int speed;  
  
 // Parameterized Constructor  
 public Car(String brand, int speed) {  
 this.brand = brand;  
 this.speed = speed;  
 }  
  
 // Copy Constructor  
 public Car(Car other) {  
 this.brand = other.brand;  
 this.speed = other.speed;  
 }  
  
 public void displayInfo() {  
 System.*out*.println("Brand: " + brand + ", Speed: " + speed);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Car originalCar = new Car("BMW", 200);  
 Car copiedCar = new Car(originalCar); // Copy constructor is called  
  
 originalCar.displayInfo();  
 copiedCar.displayInfo();  
 }  
}

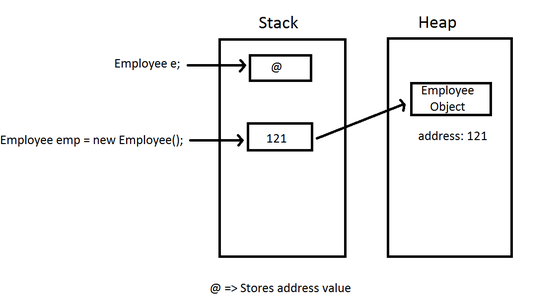
The copy constructor typically performs a **shallow copy**, meaning it only copies the values of the fields as they are. If a field is a reference type (e.g., an array or another object), the reference is copied, not the actual object. To achieve a **deep copy**, where nested objects are also copied, you need to explicitly create new instances of the nested objects.

Eg for deep copy:-

// Deep copy constructor  
public Person(Person other) {  
 this.name = other.name;  
 this.address = new Address(other.address.street, other.address.city); // New Address object for deep copy  
}

Memory Allocation for new Object

In Java, when you create an object using the new keyword, it's allocated memory on the **heap**. While variable will hold the address of memory location. Memory for object is dynamically allocated during runtime (opposite to static memory allocation for primitive data types). This is all handled by JVM.



**Stack Memory:** This is where primitive data types (like int, char, boolean, double, etc.) and method variables are stored. Stack memory is allocated and deallocated as methods are called and returned, making it efficient for managing temporary data.

**Heap Memory:** This is where objects and their instance variables are stored. Unlike stack memory, heap memory is managed by the garbage collector, which automatically reclaims memory that is no longer in use.

**Key Points:**

* Primitive types store their actual values directly on the stack.
* Objects are stored on the heap, while references to those objects are stored on the stack.
* Accessing data on the stack is faster than accessing data on the heap.
* The size of the stack is limited, while the heap can grow dynamically.

class Car{  
 int speed;  
 String color;  
  
 public Car(int speed, String color){  
 this.speed = speed;  
 this.color = color;  
 }  
}  
  
class LearnClass{  
 public static void main(String args[]){  
 Car alto = new Car(80,"Black");  
 Car suzuki = new Car(80,"Black");  
 Car maruti = alto;  
  
 //For same memory location both are equal  
 System.*out*.println(alto==maruti); //true  
 //For different memory location even with same data, this will return false  
 System.*out*.println(alto==suzuki); //false  
 }  
}

### # Destructor

In other languages (e.g., C++), a destructor is a special method that is automatically called when an object goes out of scope or is explicitly deleted. Its purpose is to clean up resources (e.g., memory, file handles, network connections) used by an object before it is removed from memory.

**Java don’t have destructors**. Java’s memory management is handled by the Java Garbage Collector (GC), which automatically frees memory for objects that are no longer reachable or used by the program.

class Resource implements AutoCloseable {  
 public void doSomething() {  
 System.*out*.println("Using the resource...");  
 }  
  
 @Override  
 public void close() {  
 System.*out*.println("Resource cleaned up.");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 try (Resource resource = new Resource()) {  
 resource.doSomething();  
 }  
 // Resource will be closed automatically at the end of this block  
 }  
}

### # Static Keyword

It is used to define members that belong to the class itself, rather than to specific instances of the class.

The static keyword belongs to the class rather than an instance of the class

When to Use static:

* Methods That Don't Require Instance Data: If a method doesn't need to access instance-specific data, make it static.
* Utility Methods: For methods that provide general utility functions, make them static.
* Shared memory allocation: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.

Real-world Example:

* **Building Example:** A building has many floors, but all the floors share the same elevator system. The elevator system is like a static variable, because it's shared by everyone in the building, no matter which floor they're on.
* **Employee Example:** A company has many employees, but they all have to follow the same company policies. These policies are like static variables, because they apply to every employee, regardless of their specific job role.

1. Static Variable

Static variables are shared across all instances of a class, meaning that only one copy of a static variable exists, regardless of how many objects are created.

Also can access static variable via class name, even without creating the object.

class BankAccount {  
 private String owner;  
 static int *totalAccounts* = 0; // Static variable shared by all instances  
  
 public BankAccount(String owner) {  
 this.owner = owner;  
 *totalAccounts*++; // Increment every time a new account is created  
 }  
  
 public void displayTotalAccounts() {  
 System.*out*.println("Total Bank Accounts: " + *totalAccounts*);  
 }  
}  
  
public class LearnStatic {  
 public static void main(String[] args) {  
 //Can access static variable without even creating the object  
 System.*out*.println(BankAccount.*totalAccounts*); //0  
   
 BankAccount account1 = new BankAccount("Alice");  
 BankAccount account2 = new BankAccount("Bob");  
  
 // Display total accounts created (same for all instances)  
 account1.displayTotalAccounts(); // Total Bank Accounts: 2  
 account2.displayTotalAccounts(); // Total Bank Accounts: 2  
 System.*out*.println(BankAccount.*totalAccounts*); //2  
 }  
}

2. Static Methods

Static variables are shared across all instances of a class, meaning that only one copy of a static variable exists, regardless of how many objects are created.

class MathUtils {  
 public static int square(int number) {  
 return number \* number;  
 }  
}  
  
public class LearnStaticMethod {  
 public static void main(String[] args) {  
 // Calling static method without creating an instance of MathUtils  
 int result = MathUtils.*square*(5);  
 System.*out*.println("Square of 5: " + result);  
 }  
}

Notes:-

* **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.
* **Can be overloaded, but not overridden:** Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

3. Static Blocks

Static blocks are executed when the class is loaded, before any objects are created or static methods are called. They’re often used for initializing static variables or performing setup tasks that need to happen only once.

class ConfigLoader {  
 static String *config*;  
  
 // Static block to initialize config data  
 static {  
 System.*out*.println("Loading configuration...");  
 *config* = "Default Configuration"; // Load configuration data  
 }  
  
}  
  
public class LearnStaticBlock {  
 public static void main(String[] args) {  
 ConfigLoader c = new ConfigLoader();  
 ConfigLoader c2= new ConfigLoader();  
 }  
}  
//Output: Loading Configuration... (printed only once)

Static block and static variables are executed in the order they are present in a program

class Test  
{  
 // static variable  
 static int *a* = *m1*(); // first this is called, thus first m1 will run  
  
 // static block (will run 2nd according to flow)  
 static {  
 System.*out*.println("Inside static block");  
 }  
  
 // static method  
 static int m1() {  
 System.*out*.println("from m1");  
 return 20;  
 }  
  
 // static method(main !!) (will run at last according to flow of program)  
 public static void main(String[] args)  
 {  
 System.*out*.println("Value of a : "+*a*);  
 System.*out*.println("from main");  
 }  
}  
//output = from m1  
//Inside static block  
//Value of a : 20  
//from main

4. Static Nested Classes

Static nested classes can be defined inside a class. Unlike inner classes, static nested classes can exist independently of an instance of the outer class. They’re useful when the nested class does not need to access instance data of the outer class.

class Outer {  
 int instanceVar = 10;  
  
 // Static nested class  
 static class InnerHelper {  
 static void display() {  
 System.*out*.println("Static nested class in action!");  
 }  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Calling static nested class method without creating an Outer instance  
 Outer.InnerHelper.*display*();  
 }  
}

InnerHelper is scoped inside Outer class, so for 1 instance of Outer, static InnerHelper is crated only once. But we can create multiple instances of InnerHelper.

class Outer {  
  
 static class InnerHelper {  
 String name;  
  
 public InnerHelper (String name) {  
 this.name = name;  
 }  
 }  
}  
  
public class TestNested {  
 public static void main(String[] args) {  
 Outer.InnerHelper a = new Outer.InnerHelper("Alice");  
 Outer.InnerHelper b = new Outer.InnerHelper("Bob");  
  
 System.*out*.println(a.name);  
 System.*out*.println(b.name);  
 }  
}  
//Output = Alice  
// Bob

**Real-World Use Cases of Static Keyword**

* Singleton Design Pattern: Static fields and methods are essential in implementing a singleton pattern, where only one instance of a class should exist.
* Utility and Helper Classes: Classes like Math, Collections, or Arrays in Java use static methods since they provide general-purpose utilities that don’t require an object instance.
* Global Counters or Constants: Static variables are often used for global constants or counters shared across instances.
* Database Connection Pooling: In many applications, a static block is used to initialize a database connection pool once, which can then be used by all instances.

## **3. Encapsulation**

Encapsulation is a fundamental principle in object-oriented programming (OOP) that combines data (attributes) and behavior (methods) in a single unit, typically a class. It also restricts direct access to the internal state of an object, allowing it to be accessed only through controlled interfaces (like methods).

How to Achieve Encapsulation in Java

* Declare fields as private to restrict direct access.
* Provide public getter and setter methods to allow controlled access to these fields.

### # Access Modifiers

Access modifiers control the visibility of classes, methods, and fields. Java has four main access levels:

* **Public:** Accessible from anywhere.
* **Private**: Accessible only within the same class.
* **Protected**: Accessible within the same package and subclasses.
* **Default** (Package-Private): Accessible only within the same package. (No keyword needed.)

Summary Table of Access Modifiers:

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |
| **public** | Yes | Yes | Yes | Yes |
| **protected** | Yes | Yes | Yes | No |
| **default** | Yes | Yes | No | No |
| **private** | Yes | No | No | No |

Use case: Directly accessing a field allows unrestricted access, which can lead to invalid or unsafe data states. Getters/setters enable controlled access and validation, protecting the internal state and ensuring data integrity.

* **Note:** Only one class in the file can be declared as public.
* The name of the file must match the name of the public class.

**Access modifiers in packages:** In Java, default (package-private) access modifiers restrict access to classes, methods, and fields to the same package only. If there is a nested package structure, the parent package and child package are treated as completely different and independent packages. Therefore:

* Default (package-private) classes in the parent package are NOT accessible by the child package.
* Default (package-private) classes in the child package are NOT accessible by the parent package.

***Java does not treat nested packages as hierarchically related***. Each package is considered distinct and separate, regardless of their naming convention.

### # Getters and Setters

Getters and setters are methods that provide controlled access to private fields:

* **Getter Method**: Used to retrieve a private field’s value.
* **Setter Method**: Used to modify a private field’s value.

By using getters and setters, you can implement validation, control, and modify behavior when accessing or updating a field.

Example:

import org.jetbrains.annotations.NotNull;  
import org.jetbrains.annotations.Range;  
  
public class SmartPhone {  
 private String phoneNo;  
 private int cost;  
  
 public String getPhoneNo() {  
 return phoneNo;  
 }  
  
 public void setPhoneNo(@NotNull String phoneNo) {  
 if(phoneNo.length()<10){  
 System.*out*.println("Phone Number is invalid, Enter again");  
 this.phoneNo = null;  
 }  
 else {  
 this.phoneNo = phoneNo;  
 }  
 }  
  
 public int getCost() {  
 return cost;  
 }  
  
 public void setCost(@Range(to = 1000, from = 100000) int cost) throws Exception {  
 if(cost>100000){  
 throw new Exception("Not possible to have this cost");  
 }  
 this.cost = cost;  
 }  
}

public class LearnEncapsulation {  
 public static void main(String[] args) throws Exception {  
 SmartPhone iPhone = new SmartPhone();  
 iPhone.setPhoneNo("12"); //Phone Number is invalid, Enter again  
 iPhone.setPhoneNo("1342524193");  
 System.*out*.println(iPhone.getPhoneNo()); //1342524193  
  
 iPhone.setCost(2125434300);  
 //Exception in thread "main" java.lang.Exception: Not possible to have this cost  
   
 System.*out*.println(iPhone.getCost());  
 }  
}

## **4. Inheritance**

Inheritance is a fundamental concept in OOP that allows a class (subclass/child class) to inherit properties and behaviors (fields and methods) from another class (superclass/parent class). It promotes code reusability and establishes a parent-child relationship between classes.

It is a IS-A relationship, also called parent-child relationship.

//Syntax  
class Parent {  
 // parent class code  
}  
class Child extends Parent {  
 // child class code  
}

**Features of Inheritance:**

* The child class inherits all non-private fields and methods of the parent class.
* Constructors are not inherited, but the child class can invoke them using the super keyword.
* Java enforces single inheritance directly to avoid ambiguity. However, multiple inheritance can be achieved using interfaces.

### # Super Keyword

Super Keyword is used to access parent class fields, methods and constructors.

package Inheritance;  
  
public class Species {  
 public int legs;  
 public int avgAge;  
 private String sound;  
 private String name;  
  
 public Species(int legs,int avgAge,String sound){  
 this.legs = legs;  
 this.avgAge = avgAge;  
 this.sound = sound;  
 }  
  
 public void setName(String name){  
 this.name = name;  
 }  
  
 public void sayHi(){  
 System.*out*.println("Hi from " + this.name);  
 }  
  
}

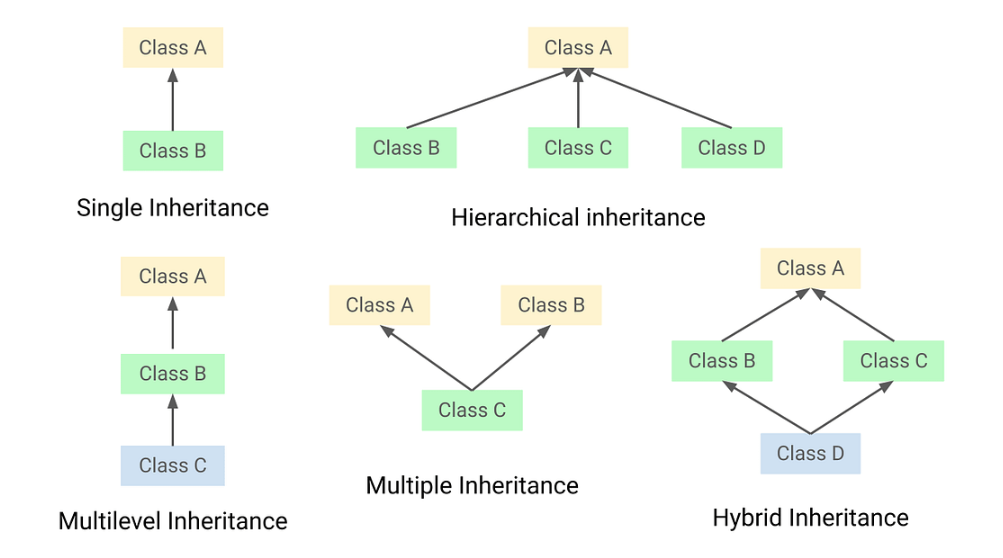
public class Dog extends Species{  
 public String name;  
  
 public Dog(String name) {  
 this.name=name;  
 super(4, 12, "Bow Bow");  
 }  
  
 public void greet(){  
 super.setName(this.name);  
 super.sayHi();  
 }  
  
 @Override  
 public String toString() {  
 return "Dog{" +  
 "name='" + name + '\'' +  
 ", legs=" + legs +  
 ", avgAge=" + avgAge +  
 '}';  
 }  
   
 public void parentData(){  
 System.*out*.println(super.legs);  
 System.*out*.println(super.avgAge);  
 //System.out.println(super.sound); is private and can't be called  
 }  
}

public class temp {  
 public static void main(String[] args) {  
 Dog champ = new Dog("Champ");  
 champ.greet(); //Hi from Champ  
 System.*out*.println(champ); //Dog{name='Champ', legs=4, avgAge=12}  
 }  
}

**Super constructor**

The Java compiler automatically inserts a call to the parent class's no-argument constructor (super()). If the parent class does not have a no-argument constructor, a compilation error occurs.

### # Types of Inheritance



**Note:**

Java avoids direct multiple inheritance to prevent the Diamond Problem, where ambiguity arises if two parent classes have methods with the same signature. This is resolved in Java by using interfaces.

Ex of multilevel inheritance:-

class Vehicle {  
 void start() {  
 System.*out*.println("Vehicle starts");  
 }  
}  
  
class Car extends Vehicle {  
 void drive() {  
 System.*out*.println("Car drives");  
 }  
}  
  
class SportsCar extends Car {  
 void accelerate() {  
 System.*out*.println("SportsCar accelerates");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 SportsCar sportsCar = new SportsCar();  
 sportsCar.start(); // Vehicle's method  
 sportsCar.drive(); // Car's method  
 sportsCar.accelerate(); // SportsCar's own method  
 }  
}

### # Method Overriding

It occurs when a subclass provides its implementation for a method that is already defined in the superclass.

Rules:

* The method must have the same name, return type, and parameters.
* The access level cannot be more restrictive than the parent class method.
* The overridden method can use the @Override annotation (optional but recommended)

class Vehicle{  
 public void sound(){  
 System.*out*.println("Vehicle make random sound");  
 }  
}  
  
class Bike extends Vehicle{  
 @Override  
 public void sound(){  
 System.*out*.println("Bike sound is Brum Brum....");  
 }  
}  
  
public class LearnOverride {  
 public static void main(String[] args) {  
 Bike bike = new Bike();  
 bike.sound(); //Bike sound is Brum Brum....  
 }  
}

## **4. Interview Questions**

**1. Are java constructors a function?**

In Java, a constructor is not considered a "function" or "method" in the traditional sense, though it is similar in some ways. Here’s why a constructor is unique and different from regular methods:

* Constructors can’t have any return type even not void.
* Constructors must have name as class.
* Constructors are automatically invoked when an object is created using the new keyword. You don’t call a constructor explicitly the way you would with a regular method.

**2. How much memory does class occupy?**

Classes do not occupy any memory. They are just the blueprint on which objects are created. Now when objects are created, they actually initialize the class members and methods and thus object consume memory.

**3. Why can’t a static method access instance variables or methods directly?**

Static methods belong to the class, not to any specific instance, so they don’t have access to instance-specific data.

**4. Why in one java file we can only have one public class?**

In Java, you can technically have more than one class in a single .java file, but only one of those classes can be declared public. This is because of how the Java compiler and runtime environment work.

Also the name of public class should be same as file name(including case).

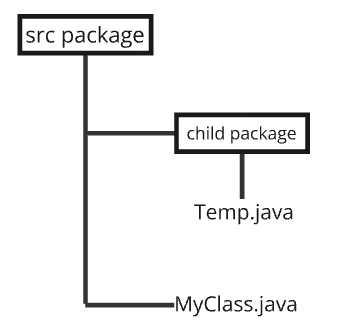
public class MyClass {  
 // code here  
}

This class must reside in a file named MyClass.java. If it doesn’t, the compiler will throw an error.

The reason for this restriction is to maintain clarity and consistency. Java assumes that if you're accessing MyClass externally (e.g., from another file), it can be found in MyClass.java.

In the same .java file, though you can define multiple non-public classes (helper classes), but they cannot be directly accessed from outside their package (also known as package-private). For example:

class TempClass{  
 //can be accessed only inside the package  
}  
public class MyClass {  
 // code here  
}



Even in given case, if MyClass.java is having a non-public class, Temp.java

can’t access that. And same if Temp.java is having non-public class, that

will not be accessible by MyClass.java

**3. Why java don’t support multiple inheritance**