**Java OOPs**

**─**

Contents

[**1. What is OOPs?** 2](#_Toc195196656)

[# Benefits of OOPs 2](#_Toc195196657)

[# Main features of OOPs 2](#_Toc195196658)

[**2. Basic Terminologies** 3](#_Toc195196659)

[# Class 3](#_Toc195196660)

[# Attributes and methods 3](#_Toc195196661)

[# Constructor 4](#_Toc195196662)

[# Destructor 9](#_Toc195196663)

[# Static Keyword 10](#_Toc195196664)

[**3. Encapsulation** 14](#_Toc195196665)

[# Access Modifiers 14](#_Toc195196666)

[# Getters and Setters 15](#_Toc195196667)

[# Final keyword 16](#_Toc195196668)

[**4. Inheritance** 18](#_Toc195196669)

[# Super Keyword 18](#_Toc195196670)

[# Types of Inheritance 20](#_Toc195196671)

[# Method Overriding 21](#_Toc195196672)

[# Interface 21](#_Toc195196673)

[# Importance of Interface 25](#_Toc195196674)

[# Default methods in interface 27](#_Toc195196675)

[# Different Cases of Inheritance 29](#_Toc195196676)

[**5. Abstraction** 30](#_Toc195196677)

[# Abstract Class 30](#_Toc195196678)

[# Abstract Methods 32](#_Toc195196679)

[**6. Polymorphism** 33](#_Toc195196680)

[**7. Advance Topics** 35](#_Toc195196681)

[# Inner Class 35](#_Toc195196682)

[# Anonymous Class 38](#_Toc195196683)

[**8. Interview Questions** 39](#_Toc195196684)

## **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**

### # 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;  
  
 // 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();  
 }  
}

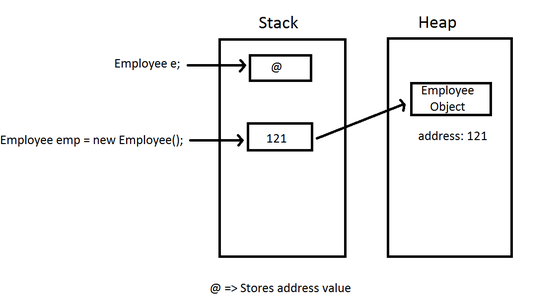
The copy constructor typically performs a **shallow copy**, meaning it only copies the values of the fields as they are. If a field is a reference type (e.g., an array or another object), the reference is copied, not the actual object. To achieve a **deep copy**, where nested objects are also copied, 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 methods can only access static variable, they can’t access instance variables as they required instance (object) but static methods are independent of object.

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. Called even before main method.

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 behaviour (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());  
 }  
}

### # Final keyword

The final keyword in Java is used to indicate that a class, method, or variable cannot be modified after they are defined. It can be applied to:

* **Variables**: Ensures the value is constant after initialization.
* **Methods**: Prevents the method from being overridden in subclasses.
* **Classes**: Prevents the class from being subclassed.
* **Constructors**: Cannot use final keyword here.

**1. Final Variable:** A final variable can be initialized only once, either at the time of declaration or in the constructor. Once a final variable is assigned a value, it cannot be changed.

It is not necessary to initiate when declared.

public class LearnFinal {  
 final int MAX\_VALUE = 100; // Final variable  
 final int MIN\_VALUE;  
  
 public LearnInheritance(int minValue) { //Can initiate with constructor  
 MIN\_VALUE = minValue;  
 }  
  
 void display() {  
 final String name; //can just declare too  
 name = "Raj";  
   
 // MAX\_VALUE = 200; // Compile-time error: cannot assign a value to a final variable  
 System.*out*.println("Max Value: " + MAX\_VALUE);  
 }  
}

**2. final method:** A final method cannot be overridden by subclasses. This ensures that the method's implementation is preserved and cannot be changed in derived classes.

class Parent {  
 final void display() {  
 System.*out*.println("Parent Display");  
 }  
}  
  
class Child extends Parent {  
 // void display() { // Compile-time error: cannot override final method  
 // System.out.println("Child Display");  
 // }  
}

**3. final class** : A final class cannot be subclassed. This is useful when you want to make sure that the behavior of a class is not altered by inheritance.

final class ImmutableClass {  
 int value;  
  
 ImmutableClass(int value) {  
 this.value = value;  
 }  
  
 public int getValue() {  
 return value;  
 }  
}  
  
// class ExtendedClass extends ImmutableClass { // Compile-time error: cannot subclass final class  
// }

**4. final Constructors :** While constructors cannot be final (since constructors are not inherited), the use of final in a class ensures that the class cannot be subclassed, indirectly preventing subclass-specific constructor modifications.

**Why final constructors don't make sense:**

Constructors are used for initialization and are not inherited. Therefore, there is no point in making a constructor final. Constructors already act as final.

Early and late binding

Binding in Java refers to the process of associating a method call with the method definition.

* **Early Binding (Static Binding):** Occurs at compile time. The method to be invoked is determined at compile time based on the reference type.

final methods, private methods, and static methods use early binding.

* **Late Binding (Dynamic Binding):** Occurs at runtime. The method to be invoked is determined at runtime based on the object type.

Overridden methods (not final) use late binding.

## **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);  
 }  
  
}

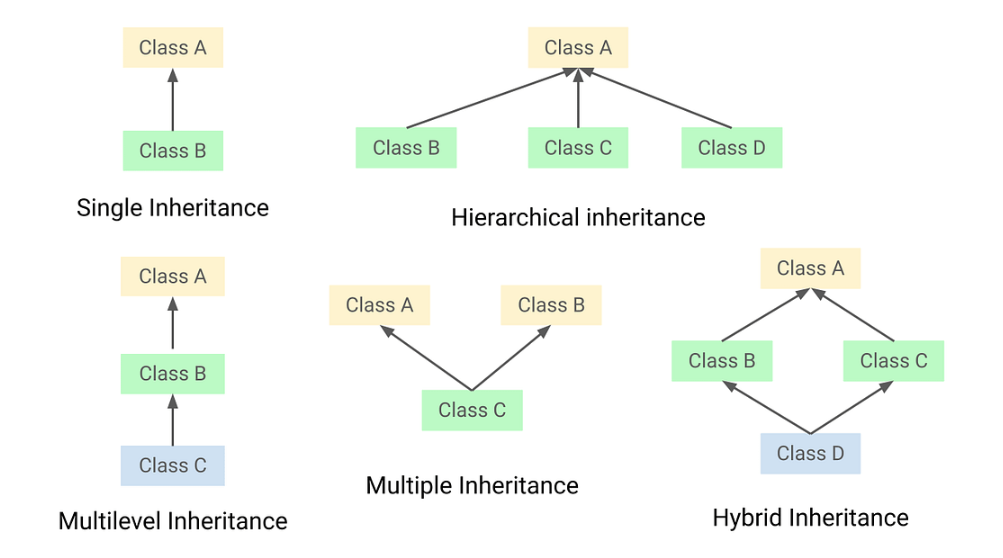
public class Dog extends Species{  
 public String name;  
  
 public Dog(String name) {  
 this.name=name;  
 super(4, 12, "Bow Bow");  
 }  
  
 public void greet(){  
 super.setName(this.name);  
 super.sayHi();  
 }  
  
 @Override  
 public String toString() {  
 return "Dog{" +  
 "name='" + name + '\'' +  
 ", 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....  
 }  
}

### # Interface

An interface in Java is a blueprint of a class. It contains abstract methods (methods without a body) that must be implemented by the classes that "implement" the interface.

Interface can also include:

* **Default methods**: Methods with a body and implementation.
* **Static methods**: Methods associated with the interface itself. From Java 9 onwards:
* **Private methods**: For encapsulating helper methods within the interface.

**Key Characteristics of Interfaces:**

* An interface cannot have instance fields (only constants).
* All methods are implicitly public and abstract (except for default, static, and private methods).
* A class can implement multiple interfaces, enabling multiple inheritance.

//Syntax  
interface InterfaceName {  
 // Constant (implicitly public, static, and final)  
 int *CONSTANT* = 100;  
  
 // Abstract method (implicitly public and abstract)  
 void abstractMethod();  
  
 // Default method  
 default void defaultMethod() {  
 System.*out*.println("Default implementation");  
 }  
  
 // Static method  
 static void staticMethod() {  
 System.*out*.println("Static method in interface");  
 }  
}

**Types of Interfaces:-**

1. **Marker Interface**:
   * An interface with no methods or fields.
   * Used to provide metadata or tagging information to classes.
   * **Example**: Serializable, Cloneable.
2. **Functional Interface**:
   * An interface with exactly one abstract method.
   * Can be implemented using **lambda expressions**.
   * **Example**: Runnable, Comparator.
   * Annotated with @FunctionalInterface.
3. **Normal Interface**:
   * An interface with multiple abstract and default methods.
   * **Example**: Collection, List.
4. **Nested Interface**:
   * An interface defined inside a class or another interface.
   * Mostly used for logical grouping.

**Use Cases:-**

1. **Achieving Abstraction**:

* Interfaces define "what" a class must do, not "how" it does it.
* Example: Vehicle interface with a move() method.

2. **Multiple Inheritance**:

* Classes can implement multiple interfaces to achieve functionality from different sources.
* Example: Flyable and Swimmable.

3. **Decoupling Code**:

* Interfaces act as a contract between components, making code modular and easier to test.

4. **Functional Programming**:

* Using functional interfaces and lambda expressions for concise and readable code.

Ex:- Class uses implements keyword to use Interface. (Note: only abstract classes can use interface without implementing all its methods.)

interface Vehicle {  
 void move(); // Abstract method  
 void run();  
}  
  
class Car implements Vehicle {  
 //Car must implement all methods of interface  
 @Override  
 public void run() {  
 System.*out*.println("Car is running");  
 }  
   
 @Override  
 public void move() {  
 System.*out*.println("Car is moving");  
 }  
}  
  
public class LearnInterface {  
 public static void main(String[] args) {  
 Vehicle myCar = new Car();  
 myCar.move(); // Output: Car is moving  
 }  
}

Ex2:- Interface with static & default methods

interface Gadget {  
 void start(); // Abstract method  
  
 default void info() {  
 System.*out*.println("This is a gadget");  
 }  
  
 static void guide() {  
 System.*out*.println("Use gadgets responsibly");  
 }  
}  
  
class Phone implements Gadget {  
 public void start() {  
 System.*out*.println("Phone is starting");  
 }  
}  
  
public class LearnInterface {  
 public static void main(String[] args) {  
 Phone phone = new Phone();  
 phone.start();  
 phone.info(); //phone able to use interface methods  
 Gadget.*guide*(); //is static methods  
 //phone.guide(); will give error  
 }  
}

Ex3:- Multiple interfaces

interface Gadget {  
 void start(); // Abstract method  
}  
  
interface Device{  
 int getPrice();  
}  
  
class Phone implements Gadget, Device{  
 int price;  
  
 public void start() {  
 System.*out*.println("Phone is starting");  
 }  
  
 @Override  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int price) {  
 this.price = price;  
 }  
}  
  
public class LearnInterface {  
 public static void main(String[] args) {  
 Phone phone = new Phone();  
 phone.start();  
 phone.setPrice(2000);  
 System.*out*.println(phone.getPrice());//2000  
 }  
}

Ex4:- Interface extending other interface

interface Gadget {  
 void start(); // Abstract method  
}  
interface Device extends Gadget{  
 int getPrice();  
}  
class Phone implements Device{  
 int price=2000;  
   
 //must implement method of Gadget too  
 public void start() {  
 System.*out*.println("Phone is starting");  
 }  
  
 @Override  
 public int getPrice() {  
 return price;  
 }  
  
}  
public class LearnInterface {  
 public static void main(String[] args) {  
 Phone phone = new Phone();  
 phone.start();  
 System.*out*.println(phone.getPrice());//2000  
 }  
}

**Key points:-**

1. Interfaces can extend multiple interfaces using a comma-separated list:

interface ChildInterface extends ParentInterface1, ParentInterface2 {

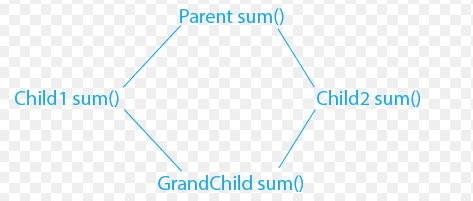
// Abstract methods

}

1. The implementing class must provide implementations for all methods inherited from all parent interfaces.

Why Java don’t allow to extend multiple class, but allow use of multiple Interface

**Diamond Problem**: Multiple inheritance can lead to the "diamond problem," where a class inherits two methods with the same signature from different parent classes. This can cause ambiguity and conflicts.



As in above case, GrandChild can’t decide which sum() method it will be using when we do GrandChild.sum().

How interface solves the issue: since interfaces just provide template and not the implementation, the implementation of code must reside in the actual class which will have only one method of same method signature.

### # Importance of Interface

Interfaces facilitate loose coupling by defining contracts (methods and properties) without specifying implementations, allowing different classes to interact through a common abstraction, minimizing dependencies and promoting flexibility

Let take example of tight coupling between bike and tyre, in given scenario bike is tightly coupled with streetTyre, so when I try to switch to dirtTyre, I will have to change a lot of code inside Bike

public class StreetTyre {  
 public void rotate(){  
 System.*out*.println("Street tyres are rotating");  
 }  
}

public class Bike {  
  
 private StreetTyre streetTyre;  
  
 public Bike(StreetTyre streetTyre) {  
 this.streetTyre = streetTyre;  
 }  
  
 public void move(){  
 streetTyre.rotate();  
 }  
}

//Example of tight coupling  
public class InterfaceDemo {  
 public static void main(String[] args) {  
 StreetTyre streetTyre = new StreetTyre();  
 Bike pulsar = new Bike(streetTyre); //Can't use dirtTyre  
 pulsar.move();   
 }  
}

Above code is tightly coupled with

public interface Tyre {  
 void rotate();  
}

public class StreetTyre implements Tyre {  
 public void rotate(){  
 System.*out*.println("Street tyres are rotating");  
 }  
}

public class DirtTyre implements Tyre {  
 public void rotate(){  
 System.*out*.println("Dirt tyres are rotating");  
 }  
}

public class Bike {  
 private Tyre tyre;  
 public Bike(Tyre tyre) {  
 this.tyre = tyre;  
 }  
 public void move(){  
 tyre.rotate();  
 }  
}

//Example of loose coupling  
public class InterfaceDemo {  
 public static void main(String[] args) {  
 Tyre streetTyre = new StreetTyre();  
 Bike pulsar = new Bike(streetTyre);  
 pulsar.move();  
  
 DirtTyre dirtTyre = new DirtTyre();  
 Bike pulsar2 = new Bike(dirtTyre);  
 pulsar2.move();  
 }  
}

### # Default methods in interface

Introduced in Java 8, default methods in interfaces allow the inclusion of method implementations. This was a significant change, as previously, interfaces could only have abstract methods.

**Notes:-**

* Default methods are allowed only in interfaces, not in abstract or regular classes.
* Static methods in interfaces are not related to default methods; they belong to the interface itself, not an instance.
* Default methods can be overridden by the implementing class.
* Use InterfaceName.super.methodName() to call a specific default method from a parent interface.

**Why Were Default Methods Introduced?**

* Backward Compatibility: To allow the addition of new methods to existing interfaces without breaking implementations of those interfaces.
* Enhancing Functional Programming: Default methods made interfaces more versatile for stream and functional programming constructs.

interface ExampleInterface {  
 void abstractMethod(); // Abstract method  
  
 default void defaultMethod() {  
 System.*out*.println("This is a default method in an interface.");  
 }  
}

**How Default Methods Differ from Regular Interface Methods?**

* Implementation: Default methods have a body (method implementation), unlike regular abstract methods.
* Overriding: Implementing classes can override default methods.

interface Vehicle {  
 void start();  
  
 default void stop() {  
 System.*out*.println("Vehicle is stopping.");  
 }  
  
 default void run() {  
 System.*out*.println("Vehicle is running.");  
 }  
}  
  
class Car implements Vehicle {  
 @Override  
 public void start() {  
 System.*out*.println("Car is starting.");  
 }  
  
 @Override  
 public void stop() {  
 System.*out*.println("Car is stopping with ABS.");  
 }  
  
 //can ignore implementing default methods  
}  
  
public class LearnInterface {  
 public static void main(String[] args) {  
 Vehicle myCar = new Car();  
 myCar.start(); // Output: Car is starting.  
 myCar.stop(); // Output: Car is stopping with ABS.  
 myCar.run(); // Output: Vehicle is running.  
 }  
}

Default methods in case of multiple inheritance

Inheritance Rules: If a class implements multiple interfaces with conflicting default methods, the implementing class must override the method explicitly to resolve ambiguity.

interface A {  
 default void show() {  
 System.*out*.println("Default method from Interface A");  
 }  
}  
  
interface B {  
 default void show() {  
 System.*out*.println("Default method from Interface B");  
 }  
}  
  
class C implements A, B {  
 @Override  
 public void show() {  
 System.*out*.println("Must fix the issue");  
 A.super.show(); // Call default method from Interface A  
 B.super.show(); // Call default method from Interface B  
 }  
}  
  
public class LearnInterface {  
 public static void main(String[] args) {  
 C obj = new C();  
 obj.show();  
 // Output: Must fix the issue  
 // Default method from Interface A  
 // Default method from Interface B  
 }  
}

### # Different Cases of Inheritance

In Java, inheritance allows a subclass (child class) to inherit properties and behaviors (fields and methods) from a superclass (parent class). However, the way we use objects and method calls depends on the reference type and object type, which can lead to different scenarios.

* **Parent obj = new Child();**: This is allowed because a Child is a Parent (substitution principle). Only methods and fields from Parent (or overridden in Child) can be accessed unless casting is used.
* **Child obj = new Child();**: This is perfectly valid. The obj reference can access both Parent and Child methods and fields.
* **Child obj = new Parent();**: This will cause a **compile-time error** because Java cannot guarantee that a Parent object is also a Child.

What all can be accessed depends on Reference type: ReferenceType obj = new ObjectType().

Case 1: Parent obj = new Child();

* Reference Type: Parent
* Object Type: Child

**What can be accessed:-**

1. **Methods and Fields from Parent**: Since the reference type is Parent, only methods and fields from the Parent class (or those overridden in Child) are accessible.
2. **Overridden Methods in Child**: If the child class overrides a method from the parent, the version from the Child class will be called (runtime polymorphism).
3. **Fields and Methods in Child**: The reference obj cannot directly access fields and methods that are specific to Child. If you want to call a method unique to Child, you'll need to cast it.

**Scope**: The reference type (Parent) determines which methods can be called. You cannot directly call methods that are exclusive to Child unless you cast obj to Child.

class Parent {  
 void display() {  
 System.*out*.println("Parent display");  
 }  
}  
  
class Child extends Parent {  
 @Override  
 void display() {  
 System.*out*.println("Child display");  
 }  
  
 void childSpecificMethod() {  
 System.*out*.println("Child specific method");  
 }  
}  
  
public class LearnInheritance {  
 public static void main(String[] args) {  
 Parent obj = new Child();  
 obj.display(); // Calls "Child display" because of method overriding  
 // obj.childSpecificMethod(); // Error! 'childSpecificMethod' is not in Parent class  
 }  
}

## **5. Abstraction**

Abstraction is the process of **hiding implementation details** and exposing only essential functionalities to the user.  
It allows developers to focus on **what an object does** rather than **how it does it**.

In Java, abstraction is achieved through:

1. **Abstract classes**
2. **Interfaces**

Use case:

1. **Reusability**: Abstract classes can define common behavior for multiple related subclasses.
2. **Extensibility**: Enables creating a flexible hierarchy of classes with shared behavior. Example: Vehicle → Car, Bike.
3. **Maintainability**: Focuses on high-level design, reducing implementation complexity.

### # Abstract Class

An abstract class is a special class in Java that is used to represent incomplete or partially implemented objects. It serves as a blueprint for other classes and cannot be instantiated directly. Abstract classes allow developers to define common behavior while leaving some methods for the derived classes to implement.

An abstract class in Java is a class that:

* Cannot be instantiated.
* Can have both abstract methods (methods without a body) and concrete methods (methods with an implementation).

abstract class Shape {  
 abstract double calculateArea();  
  
 void display() { //concrete method  
 System.*out*.println("This is a shape.");  
 }  
}  
  
class Circle extends Shape {  
 double radius;  
  
 Circle(double radius) {  
 this.radius = radius;  
 }  
 //subclass must implement abstract methods  
 @Override  
 double calculateArea() {  
 return Math.*PI* \* radius \* radius;  
 }  
}  
  
public class LearnAbstraction {  
 public static void main(String[] args) {  
 Shape shape = new Circle(5);  
 shape.display(); // Output: This is a shape.  
 System.*out*.println("Area: " + shape.calculateArea()); // Output: Area: 78.53981633974483  
 //Shape s = new Shape(); //gives error(abstract class can’t be instantiated)  
 }  
}

Constructor in abstract class

Although you cannot instantiate an abstract class directly, constructors in abstract classes are used to initialize common fields when an object is created through a subclass. Subclass must extend parent class constructor.

abstract class Animal {  
 String name;  
  
 // Constructor  
 Animal(String name) {  
 this.name = name;  
 }  
  
 abstract void makeSound();  
  
 void displayInfo() {  
 System.*out*.println("This is " + name);  
 }  
}  
  
class Dog extends Animal {  
 //Subclass must extend and run parent constructor  
 Dog(String name) {  
 super(name);  
 }  
  
 @Override  
 void makeSound() {  
 System.*out*.println("Bark");  
 }  
}  
  
public class LearnAbstraction {  
 public static void main(String[] args) {  
 Dog dog = new Dog("Buddy");  
 dog.displayInfo(); // Output: This is Buddy  
 dog.makeSound(); // Output: Bark  
 }  
}

**Difference between Interface and Abstract Class**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Purpose** | Used for partial abstraction. | Used for 100% abstraction. |
| **Methods** | Can have both abstract and concrete methods. | Can have abstract, default, static methods (from Java 8). |
| **Fields** | Can have instance variables. | Can only have public static final constants. |
| **Constructors** | Can have constructors. | Cannot have constructors. |
| **Inheritance** | Supports single inheritance. | Supports multiple inheritance. |
| **Access Modifiers** | Methods and fields can have any access modifier. | Methods are implicitly public (except default methods). |
| **Use Cases** | For shared code and common functionality. | For defining a contract with no implementation. |
| **Performance** | Slightly faster than interfaces due to inheritance. | Slightly slower due to multiple inheritance. |

### # Abstract Methods

An abstract method is a method declared without an implementation (no method body). It is meant to be overridden in subclasses that inherit from an abstract class. Abstract methods define a contract for subclasses, ensuring they provide their specific implementation.

**Characteristics: -**

1. **Declaration**:
   * Declared using the abstract keyword.
   * Does not have a method body ({}).
   * Ends with a semicolon (;).
2. **Purpose**:
   * Provides a way to define methods that must be implemented by subclasses.
   * Enforces a consistent interface in subclasses.
3. **Usage**:
   * Abstract methods **must be declared in an abstract class**.
   * Subclasses inheriting the abstract class must implement all its abstract methods unless the subclass is also abstract.
4. **Access Modifiers**:
   * Can use any access modifier (public, protected, private), but they are typically declared as public or protected to allow subclass access.
5. **Notes:**
   * An abstract method can only exist inside an abstract class(and in interface, where we don’t even need to use keyword abstract)
   * The first concrete (non-abstract) subclass must implement the abstract method.
   * Cannot Be Static or Final: Abstract methods are meant to be overridden, which conflicts with the immutability of final and the class-level nature of static.
   * Cannot Be Private: Abstract methods are designed for sub classing, so private is not allowed.

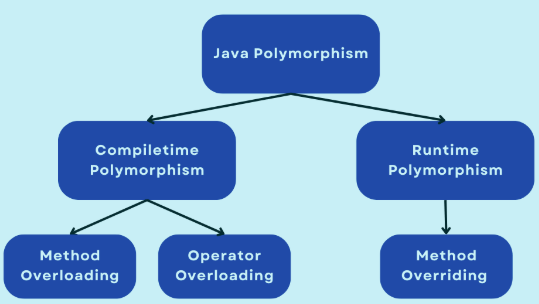
abstract class Animal {  
 String name;  
  
 //abstract method  
 abstract void makeSound();  
  
}  
  
class Dog extends Animal {  
 @Override  
 void makeSound() {  
 System.*out*.println("Bark");  
 }  
}  
  
public class LearnAbstraction {  
 public static void main(String[] args) {  
 Dog dog = new Dog();  
 dog.makeSound(); // Output: Bark  
 }  
}

## **6. Polymorphism**

Polymorphism in Java means the ability of a single method or object to behave in different ways based on the context. It is one of the core principles of object-oriented programming.

Literal Meaning: Poly = Many, Morph = Forms.

Importance: Provides flexibility and enhances code reusability.



Compile-time Polymorphism (Method Overloading)

Occurs when multiple methods in the same class have the same name but different parameters. The method to be called is resolved at compile time.

class MyCalculator {  
 // Method Overloading  
 int add(int a, int b) {  
 return a + b;  
 }  
  
 double add(double a, double b) {  
 return a + b;  
 }  
  
 int add(int a, int b, int c) {  
 return a + b + c;  
 }  
}  
  
public class LearnPolymorphism {  
 public static void main(String[] args) {  
 MyCalculator calc = new MyCalculator();  
 System.*out*.println(calc.add(5, 10)); // Calls int add(int, int)  
 System.*out*.println(calc.add(5.5, 10.2)); // Calls double add(double, double)  
 System.*out*.println(calc.add(1, 2, 3)); // Calls int add(int, int, int)  
 }  
}

­Run-time Polymorphism (Method Overriding)

Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method call is resolved at runtime.

class Animal {  
 void sound() {  
 System.*out*.println("Animal makes a sound");  
 }  
}  
  
class Dog extends Animal {  
 @Override  
 void sound() {  
 System.*out*.println("Dog barks");  
 }  
}  
  
public class LearnPolymorphism {  
 public static void main(String[] args) {  
 Animal myAnimal = new Dog(); // Upcasting  
 myAnimal.sound(); // Calls Dog's sound() at runtime  
 }  
}

## **7. Advance Topics**

### # Inner Class

An inner class is a class defined inside another class or interface. They are used to logically group classes that will be used only in one place, improve encapsulation, and allow better access to the enclosing class's members. Java provides four types of inner classes:

* Member Inner Classes: A non-static inner class defined within the body of an outer class.
* Static Nested Classes: A static class defined inside another class.
* Local Inner Classes: A class defined within a method or block.
* Anonymous Inner Classes: A class without a name that is used for quick, one-time use.

Uses of Inner class:

* Encapsulate helper classes to be used only by their enclosing class.
* Make code more readable by logically grouping classes.
* Gain access to private members of the enclosing class directly (Inner classes can access the private, protected, or public members of the enclosing (outer) class directly)
* Provide a concise way to implement interfaces or extend classes (via anonymous inner classes).

Member Inner class:

A member inner class is a non-static inner class that is defined at the same level as instance variables and methods within the enclosing class. It has access to all the members (including private ones) of the outer class.

class Outer {  
 private String message = "Hello from Outer";  
  
 // Member Inner Class  
 class Inner {  
 void displayMessage() {  
 System.*out*.println(message); // Accessing outer class's private member  
 }  
 }  
}  
  
public class LearnInnerClass {  
 public static void main(String[] args) {  
 Outer out = new Outer();  
 Outer.Inner in = out.new Inner();  
 in.displayMessage(); //Hello from Outer  
 }  
}

To create an instance of a member inner class:

* First, create an instance of the outer class.
* Use the outer class instance to create the inner class instance

Static Nested class:

A static nested class is a nested class defined with the static keyword. Unlike member inner classes, static nested classes do not require an instance of the outer class to be created.

class Outer {  
 static String *message* = "Hello from Outer";  
  
 // Static Nested Class  
 static class Inner {  
 void displayMessage() {  
 System.*out*.println(*message*); // Accessing static member of the outer class  
 }  
 }  
}  
  
public class LearnInnerClass {  
 public static void main(String[] args) {  
 Outer out = new Outer();  
 Outer.Inner in = new Outer.Inner();  
 in.displayMessage(); //Hello from Outer  
 }  
}

Local Inner class:

A local inner class is defined within a method or a block of code. It is scoped to the enclosing method or block and cannot be accessed from outside it.

class Outer {  
 void display() {  
 final int x = 10; // Must be effectively final for use in local inner class  
  
 // Local Inner Class  
 class Local {  
 void print() {  
 System.*out*.println("Value of x: " + x);  
 }  
 }  
  
 Local local = new Local(); // Create instance within the method  
 local.print(); // Output: Value of x: 10  
 }  
}  
  
public class LearnInnerClass {  
 public static void main(String[] args) {  
 Outer out = new Outer();  
 out.display(); //Value of x: 10  
 }  
}

A local inner class can access local variables of the enclosing method, but they must be effectively final (i.e., their value cannot be modified after initialization).

class Outer {  
 void display() {  
 int x = 10; // Must be effectively final for use in local inner class  
 //can use without final also, but then it should not be modified later  
  
 // Local Inner Class  
 class Local {  
 void print() {  
 System.*out*.println("Value of x: " + x);  
 }  
 }  
 x=x+1; //wrong,   
 Local local = new Local(); // Create instance within the method  
 local.print(); //Error  
 //java: local variables referenced from an inner class must be final or effectively final  
 }  
}

Anonymous Inner class:

An anonymous inner class is a class without a name that is declared and instantiated in a single statement. It is often used for quick, one-time use, such as implementing interfaces or extending classes inline

interface Greeting {  
 void sayHello();  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Anonymous Inner Class implementing the Greeting interface  
 Greeting greeting = new Greeting() {  
 @Override  
 public void sayHello() {  
 System.*out*.println("Hello, World!");  
 }  
 };  
 greeting.sayHello(); // Output: Hello, World!  
 }  
}

Anonymous inner classes are commonly used to provide implementations for interfaces or abstract classes without the need to create a separate class.

public class Main {  
 public static void main(String[] args) {  
 // Using anonymous inner class to create a thread  
 Thread thread = new Thread() {  
 @Override  
 public void run() {  
 System.*out*.println("Thread is running...");  
 }  
 };  
 thread.start();  
 }  
}

### # Anonymous Class

An anonymous class in Java is a type of inner class without a name. It is used to create a one-time implementation of an interface or a subclass of a class. Anonymous classes are widely used in situations where a simple implementation of an interface or abstract class is required, and creating a separate named class is unnecessary.

* A class declared and instantiated in a single statement.
* Cannot have a constructor because it does not have a name.
* Typically used for:

1. Implementing an interface.
2. Extending an abstract or concrete class.
3. Providing inline, concise code for short-lived use cases.

//Syntax  
InterfaceName obj = new InterfaceName() {  
 // Implementation of methods  
};

Ex:-

interface Greeting {  
 void sayHello();  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Anonymous class implementing Greeting interface  
 Greeting greeting = new Greeting() {  
 @Override  
 public void sayHello() {  
 System.*out*.println("Hello, World!");  
 }  
 };  
  
 greeting.sayHello(); // Output: Hello, World!  
 }  
}

When an anonymous class extends a class, it can override the methods of the class.

abstract class Animal {  
 abstract void sound();  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Anonymous class extending Animal  
 Animal animal = new Animal() {  
 @Override  
 void sound() {  
 System.*out*.println("Animal makes a sound");  
 }  
 };  
  
 animal.sound(); // Output: Animal makes a sound  
 }  
}

## **8. 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. Can we inherit the constructor?**

No, constructors cannot be inherited in Java. While constructors are not inherited, a subclass can call a constructor from its superclass using the super() keyword to initialize the parent class. Constructors are specific to the class in which they are defined and are not inherited as part of the subclass's method set. However, a subclass can invoke the superclass constructor explicitly or implicitly, but it cannot inherit or override the superclass constructor.

**4. Why final methods are faster than other methods?**

* **Early Binding**: Final methods are resolved at compile time, avoiding runtime method lookup, making execution faster.
* **No Overriding**: Since final methods can't be overridden, the JVM doesn't need to determine the actual object's method during runtime.
* **Optimized Execution**: The JVM can inline final methods during optimization, further improving performance.
* **Simplified Call**: Final methods eliminate the dynamic dispatch overhead associated with non-final methods.

**5. 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.

**6. 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).

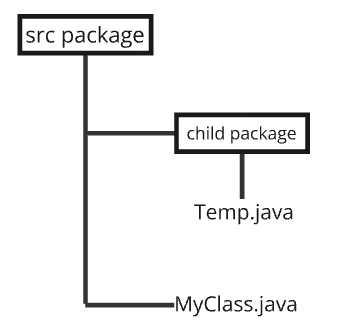
public class MyClass {  
 // code here  
}

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

**7. Why default methods were introduced to interface, what are it’s use cases.**

Before Java 8, interfaces could not have implemented methods. Adding new methods to an existing interface would break all the classes implementing it because those classes would be forced to implement the new method. This limitation made interfaces rigid and hard to evolve in APIs.

Default methods enable the addition of new methods to an interface without breaking the existing implementations. The implementing classes can choose to override the default implementation if needed.

Eg: Java's List interface got default void sort() in Java 8.

**8. Can static methods be override?**

No, static methods cannot be overridden in Java. Static methods are associated with the class rather than with an instance of the class, so they are resolved at compile-time rather than at runtime. If a subclass defines a static method with the same signature as a static method in the parent class, it hides the method rather than overriding it. This is called method hiding, and the method called is determined by the reference type, not the object type, which contrasts with regular method overriding where the method called is determined by the object type.

**9. Why Method Overloading is Not Possible by Changing the Return Type Only?**

Method overloading cannot be achieved by merely changing the return type of the method. This is because the return type alone is not sufficient to distinguish between two methods. The Java compiler will not be able to resolve the call to the appropriate method based on return type alone, leading to ambiguity.

**10. Can an abstract class have a final method?**

Yes, an abstract class can have final methods. These methods cannot be overridden by subclasses but can be inherited.

**11. Why can't abstract methods be static in Java?**

Abstract methods are meant to be overridden by subclasses, while static methods are bound to the class and cannot be overridden.

**12. Is it Necessary for an Abstract Class to Have Abstract Methods?**

No, it is **not necessary** for an abstract class to have abstract method.  
An abstract class can exist without any abstract methods. However, such classes are still declared abstract because they are not intended to be instantiated directly. Abstract classes without abstract methods can serve as a **base class** to provide common functionality for subclasses.

**13. Why Don’t We Need the** abstract **Keyword for Abstract Methods in an Interface?**

**Reason**: In an interface, all methods are **implicitly abstract** (before Java 8) because the purpose of an interface is to provide a contract that specifies method signatures without implementation. Thus, there is no need to explicitly use the abstract keyword for methods in an interface.

**Comparison with Abstract Class**:  
Abstract methods in abstract classes require the abstract keyword because the class itself can have **concrete methods**. The keyword helps distinguish which methods are abstract and must be implemented by subclasses.

**14. Why can’t we have abstract constructor?**

In Java, an abstract class can have a constructor, but it cannot have an abstract constructor. A constructor in an abstract class is used to initialize common properties for subclasses, but since an abstract class cannot be instantiated directly, its constructor is invoked when a subclass is instantiated, typically using the super() keyword. The purpose of a constructor is to initialize an object, and abstract methods serve as a blueprint for subclasses to provide implementations. Abstract constructors would be meaningless because constructors are not inherited and cannot be implemented in subclasses, making the concept of an abstract constructor illogical.

**15. What are limitations of Inheritance?**

Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also classes involved in inheritance are tightly couples, so change in one needs to be nested in all connected classes.

**16. What is coupling in OOPs?**

Coupling refers to degree of dependency between different classes or modules, it indicates how much one class or module relies on another, it is of 2 type:

1. Loose coupling: Classes are independent and change in one class are less likely to affect another.

2. Tight coupling: Classes are highly dependent on each other and changes in one class are likely to affect other.

**17. What is composition in OOPs?**

Composition in Java is a design principle where one class contains an instance of another class, enabling a "has-a" relationship. It is used to achieve code reuse and flexibility.

class Engine {  
 void start() {  
 System.*out*.println("Engine starts");  
 }  
}  
  
class Car {  
 private Engine engine; // Composition: Car "has-a" Engine  
  
 Car() {  
 this.engine = new Engine(); // Car contains an Engine instance  
 }  
  
 void drive() {  
 engine.start(); // Using Engine's functionality  
 System.*out*.println("Car is driving");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Car car = new Car();  
 car.drive();  
 }

}

**18. What is lazy initialization?**

Lazy initialization is a technique where an object or resource is initialized only when it is needed, rather than at the time of object creation. This is useful for saving memory and improving performance if the resource is expensive to initialize. Eg: use of getInstance() in singleton classes

class LazyExample {  
 private String data; // Data to be lazily initialized  
  
 // Getter for lazy initialization  
 public String getData() {  
 if (data == null) { // Initialize only if it hasn't been already  
 data = "This is lazily initialized data";  
 }  
 return data;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 LazyExample example = new LazyExample();  
 System.*out*.println("Before accessing data...");  
  
 // Data is initialized here, only when accessed  
 System.*out*.println(example.getData()); // Output: This is lazily initialized data  
 }  
}

**19. What are singleton classes?**

A **Singleton** is a design pattern that ensures a class has **only one instance** throughout the application's lifecycle and provides a global point of access to that instance.

**Key Features:**

1. Restricts the creation of multiple objects.
2. Useful for shared resources like configurations, database connections, or logging mechanisms.

**Use Case:**

Singleton is commonly used for:

* **Database Connection Pool**: Ensures only one connection pool is maintained.
* **Logging Framework**: Ensures all parts of the application use the same logger instance.
* **Configuration Manager**: Holds shared application settings.

class Singleton {  
 // Static variable to hold the single instance  
 private static Singleton *instance*;  
  
 // Private constructor to prevent instantiation from other classes  
 private Singleton() {}  
  
 // Public method to provide global access to the instance  
 public static Singleton getInstance() {  
 if (*instance* == null) { // Create instance if it doesn't exist  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
  
 public void showMessage() {  
 System.*out*.println("Singleton instance in action!");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Get the singleton instance  
 Singleton singleton = Singleton.*getInstance*();  
 singleton.showMessage(); // Output: Singleton instance in action!  
 }  
}

**20. Can we overload main method?**

* Only this method is treated as the entry point by the JVM:

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

* All other overloaded main methods are just like regular static methods.
* You must manually call them; JVM won’t invoke them on program start.

Eg: *public static void main(double[] args)* can be called as *main(2.14)*