**Java OOPs**

**─**

Contents

[**1. What is OOPs?** 2](#_Toc182333572)

[# Benefits of OOPs 2](#_Toc182333573)

[# Main features of OOPs 2](#_Toc182333574)

[**2. Basic Terminologies** 3](#_Toc182333575)

[# Class 3](#_Toc182333576)

[# Attributes and methods 3](#_Toc182333577)

[# Constructor 4](#_Toc182333578)

[# Destructor 9](#_Toc182333579)

[**3. Interview Questions** 10](#_Toc182333580)

## **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();  
 }  
}

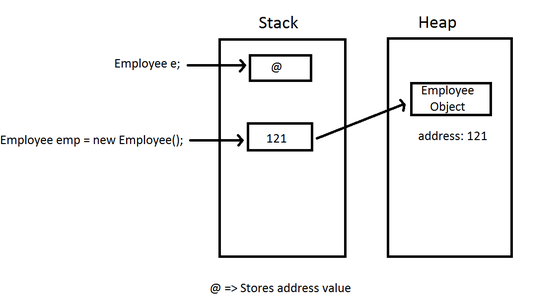
The copy constructor typically performs a **shallow copy**, meaning it only copies the values of the fields as they are. If a field is a reference type (e.g., an array or another object), the reference is copied, not the actual object. To achieve a **deep copy**, where nested objects are also copied,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

## **3. 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.