JAVA OOPS

we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as inheritance, data binding, polymorphism, etc. Simula is considered the first object-oriented programming language.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects.

Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

Coupling, Cohesion, Association, Aggregation, Composition

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If one task is performed in different ways, it is known as polymorphism. In Java, we use method overloading and method overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing. In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts.

The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects: One to One, One to Many, Many to One, and Many to Many. **For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).**

Aggregation and Composition are both forms of association in object-oriented programming, representing relationships between objects.

- Represents a weak relationship.

- One object contains other objects as part of its state.

- The contained objects can exist independently of the parent object.

Example:

A `Library` and `Book`:

- A `Library` has multiple `Book` objects.

- If the `Library` is deleted, the `Book` objects can still exist.

```
class Book {
   String title;
   // Book class code
}

class Library {
   List<Book> books;
   // Library class code
}
```

Composition

- Represents a strong relationship.

- One object contains other objects as part of its state.

- The contained objects cannot exist independently of the parent object.

- If the parent object is deleted, all the contained objects are deleted too.

Example

A `House` and `Room`:

- A `House` is composed of multiple `Room` objects.

- If the `House` is demolished, the `Room` objects cease to exist.

```
class Room {
   String name;
   // Room class code
}
class House {
   List<Room> rooms;
   // House class code
}
```

In summary, aggregation allows independent existence of contained objects, while composition does not.

What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

OBJECT - An object has three characteristics:

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

An object is an instance of a class. A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain: Fields, Methods ,Constructors, Blocks, Nested class and interface.

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class and outside the class

```java
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

```java
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Initializing an object means storing data into the object –

Initialization through reference

```java
public static void main(String args[]){
 //Creating objects
 Student s1=new Student();
 Student s2=new Student();
 //Initializing objects
 s1.id=101;
 s1.name="Sonoo";
```

Initialization through method

```java
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
```

Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Initialization through a constructor

```java
class Person {
    String name;
    int age;

    // Constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        System.out.println(person.name + " is " + person.a
    }
}
```

different ways to create an object in Java –

If you have to use an object only once, an anonymous object is a good approach. **-> new Calculation();**

Calling method through a reference **-> Calculation c=new Calculation(); c.fact(5);**

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

Public: The method is accessible by all classes when we use public specifier in our application.

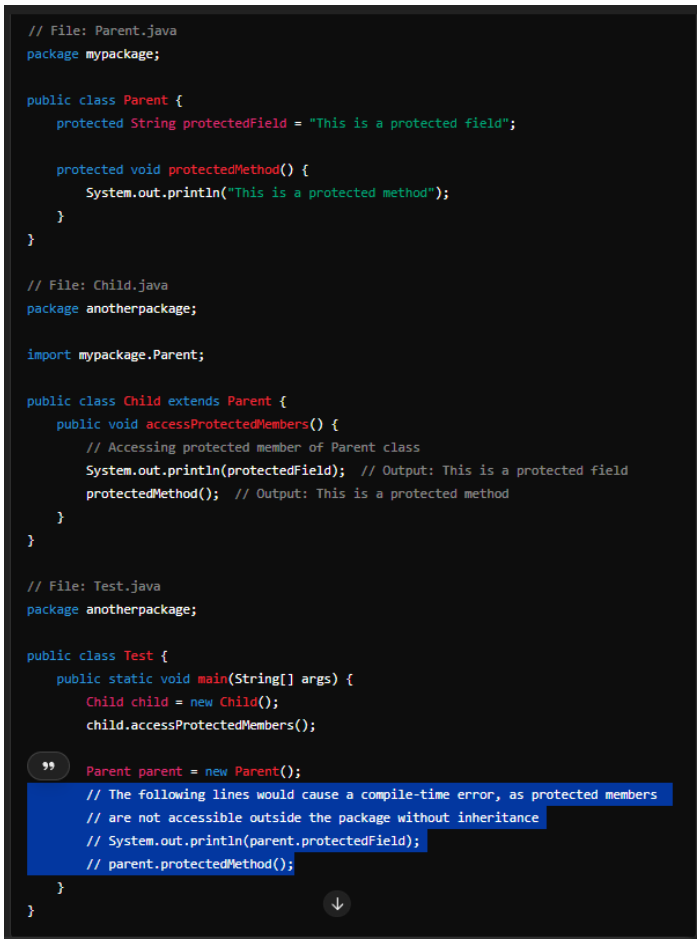
Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.

Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.

Package-Level Access: protected members are accessible within the same package, just like package-private (default) members.

Inheritance Access: protected members can be accessed in subclasses. This is true even if the subclass is in a different package, which is not allowed with the default (package-private) access modifier.

Default: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.



```java
// File: Parent.java
package mypackage;

public class Parent {
    protected String protectedField = "This is a protected field";

    protected void protectedMethod() {
        System.out.println("This is a protected method");
    }
}

// File: Child.java
package anotherpackage;

import mypackage.Parent;

public class Child extends Parent {
    public void accessProtectedMembers() {
        // Accessing protected member of Parent class
        System.out.println(protectedField);  // Output: This is a protected field
        protectedMethod();  // Output: This is a protected method
    }
}

// File: Test.java
package anotherpackage;

public class Test {
    public static void main(String[] args) {
        Child child = new Child();
        child.accessProtectedMembers();

        Parent parent = new Parent();
        // The following lines would cause a compile-time error, as protected members
        // are not accessible outside the package without inheritance
        // System.out.println(parent.protectedField);
        // parent.protectedMethod();
    }
}
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword static before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the main() method.

```
class MathUtils {
    // Static method to calculate the square of a number
    public static int square(int number) {
        return number * number;
    }
}

public class Main {
    public static void main(String[] args) {
        // Calling the static method without creating an instance of MathUtils
        int result = MathUtils.square(5);
        System.out.println("The square of 5 is: " + result);  // Output: The square of 5 i
    }
}
```

///instance method left

Constructors

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default. There are two types of constructors in Java: no-arg constructor, and parameterized constructor. **It is called constructor because it constructs the values at the time of object creation.**

What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Constructor Overloading

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

```
//creating two arg constructor
Student5(int i,String n){
id = i;
name = n;
}
```

```
//creating three arg constructor
Student5(int i,String n,int a){
id = i;
name = n;
age=a;
}
```

The constructor is invoked implicitly. The method is invoked explicitly.

Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

The static keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static can be – variable, method, block, nested class.

If you declare any variable as static, it is known as a static variable.

The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

The static variable gets memory only once in the class area at the time of class loading. It makes your program memory efficient (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

EXAMPLE -

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once. static String college ="ITS";//static variable

//Program of the Counter without static

```
class Counter {
    int count = 0; // will get memory each time

    Counter() {
        count++; // incrementing value
        System.out.println(count);
    }

    public static void main(String args[]) {
        // Creating objects
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();
    }
}
```

```
//is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

In the provided Java example, we have a class Counter that demonstrates the use of an instance variable count, which gets memory each time an object of the class is created. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable.

Now if We use static - static int count=0;//will get memory only once and retain its value
static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

```
class HelloWorld {
    public static void main(String[] args) {
        Student st = new Student("Aadi", 11);
        st.display();
        Student.change();
        st.display();
    }
}
class Student{
    static String college = "ITB";
    int roll;
    String name;
    static void change(){
        college = "BTS";
    }
    Student(String n, int roll){
        this.roll = roll;
        this.name = n;
    }
    void display(){
        System.out.println(name + " Your college is : "+ college);
    }
}
```

```
class Main {
    public static void main(String[] args) {
        int result = Calculate.cube(Calculate.b);
        System.out.println(result);
    }
}
class Calculate {
    int a = 10; // cannot call a
    static int b = 5; // but can call b directly
    static int cube(int x) {
        return x*x*x;
    }
}
```

There are two main restrictions for the static method. They are:

- The static method can not use non static data member or call non-static method directly.
- this and super cannot be used in static context.

Q) Why is the Java main method static?

- Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

Java static block -> Is used to initialize the static data member. It is executed before the main method at the time of classloading.

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

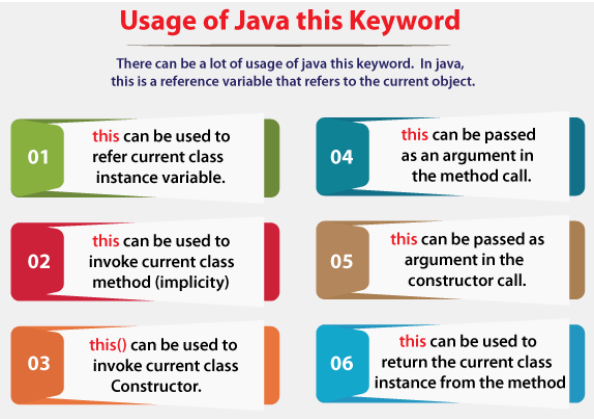
```
Output:static block is invoked
       Hello main
```

Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

THIS

There can be a lot of usage of Java this keyword. In Java, this is a reference variable that refers to the current object.



To refer current class instance variable - To refer to the current class instance variable. Resolves ambiguity between instance variables and parameters.

Without this Keyword

Example where instance variables and parameters have the same name:

```java
class Student {
    int rollno;
    String name;
    float fee;

    Student(int rollno, String name, float fee) {
        rollno = rollno;  // Ambiguity, instance variable not set
        name = name;
        fee = fee;
    }

    void display() {
        System.out.println(rollno + " " + name + " " + fee);
    }
}

class TestThis1 {
    public static void main(String args[]) {
        Student s1 = new Student(111, "ankit", 5000f);
        Student s2 = new Student(112, "sumit", 6000f);
        s1.display();  // Output: 0 null 0.0
        s2.display();  // Output: 0 null 0.0
    }
}
```

```java
class Student {
    int rollno;
    String name;
    float fee;

    Student(int rollno, String name, float fee) {
        this.rollno = rollno;  // 'this' refers to instance vari
        this.name = name;
        this.fee = fee;
    }

    void display() {
        System.out.println(rollno + " " + name + " " + fee);
    }
}

class TestThis2 {
    public static void main(String args[]) {
        Student s1 = new Student(111, "ankit", 5000f);
        Student s2 = new Student(112, "sumit", 6000f);
        s1.display();  // Output: 111 ankit 5000.0
        s2.display();  // Output: 112 sumit 6000.0
    }
}
```

Issue: Instance variables remain uninitialized due to ambiguity.

Solution with this Keyword - Using this to resolve ambiguity 2nd image.
If parameters have different names than instance variables, then it is not required.

To invoke current class method – this.method();

To invoke current class constructor –

```java
class TestThis5{
    public static void main(String args[]){
        A a=new A(10);
    }
}
class A{
    A(){System.out.println("hello a");}
    A(int a, int b){System.out.println(a + 5 + b);}
    A(int x){
        this(); // hello a
        this(4,6);// 15
        System.out.println(x); // 10
    }
}
```

Real usage of this() constructor call - The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

Constructor chaining –

```java
class Student {
    int rollno;
    String name, course;
    float fee;
    // Constructor to initialize rollno, name, and course
    Student(int rollno, String name, String course) {
        this.rollno = rollno;
        this.name = name;
        this.course = course;
    }
    // Constructor to initialize rollno, name, course, and fee
    Student(int rollno, String name, String course, float fee) {
        this(rollno, name, course);  // Reusing the first constructor
        this.fee = fee;
    }
    // Method to display student details
    void display() {
        System.out.println(rollno + " " + name + " " + course + " " + fee);
    }
}
```

```java
public class TestThis7 {
    public static void main(String args[]) {
        // Creating objects using different constructors
        Student s1 = new Student(111, "ankit", "java");
        Student s2 = new Student(112, "sumit", "java", 6000f);

        // Displaying the details of the students
        s1.display();
        s2.display();
    }
}
```

this can be passed as an argument in the method call.

this can be passed as argument in the constructor call.

This can be used to return the current class instance from the method.

```java
class Yum {
    int a = 5;
    void m(int x){
        System.out.println(this.a);
    }
}

public class HelloWorld {
    public static void main(String[] args) {
        Yum y = new Yum();
        System.out.println(y);
        y.m(5);
    }
}
```

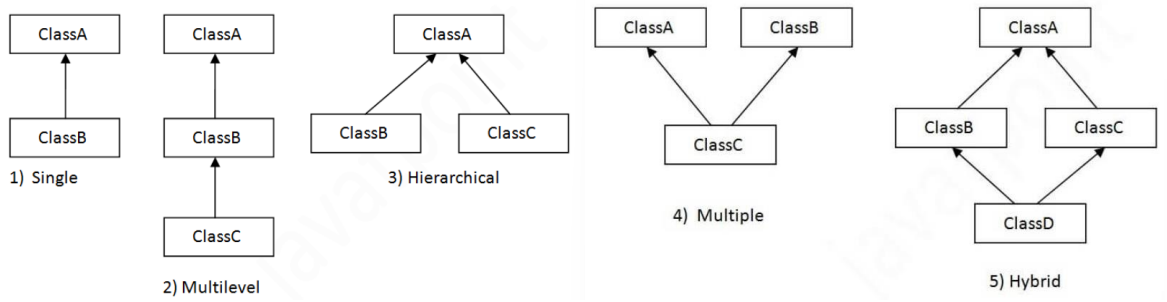
The this keyword in Java always refers to the current instance of the class. When you use this inside an instance method, it refers to the object on which the method was called, not to the method's parameters or local variables. Thus, this will always print the reference ID of the object, not the values of the method parameters or local variables.

Inheritance in Java

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also. Inheritance represents the IS-A relationship which is also known as a parent-child relationship. We use Inheritance For Method Overriding (so runtime polymorphism can be achieved). & For Code Reusability.

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality. In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass. **Programmer IS-A Employee**

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.



Multiple inheritance is not supported in Java through class.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
```

Single Level

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
```

MultiLevel

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
```

Hierarchical

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class. Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship. Consider a situation, Employee object contains many informations such as id, name, emaillId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below. Similarly ,a Library can contain multiple Book objects, representing the HAS-A relationship. Each Book has its own attributes such as title, author, and ISBN. The Library class has a collection of Book objects, showing that a library HAS-A collection of books.

```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address. Why use Aggregation? For Code Reusability.

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

This example effectively demonstrates aggregation by showing how an Employee class (represented by Emp) has a reference to an Address object. This relationship is an example of aggregation because the Address objects can exist independently of the Employee objects. Let's break down the code and its significance in detail –

Address Class

java Copy code

```java
public class Address {
    String city, state, country;

    public Address(String city, String state, String country) {
        this.city = city;
        this.state = state;
        this.country = country;
    }
}
```

```java
public class Emp {
    int id;
    String name;
    Address address;

    public Emp(int id, String name, Address address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    void display() {
        System.out.println(id + " " + name);
        System.out.println(address.city + " " + address.state
    }

    public static void main(String[] args) {
        Address address1 = new Address("gzb", "UP", "india");
        Address address2 = new Address("gno", "UP", "india");

        Emp e = new Emp(111, "varun", address1);
        Emp e2 = new Emp(112, "arun", address2);

        e.display();
        e2.display();
    }
}
```

```
111 varun
gzb UP india
112 arun
gno UP india
```

The Emp class represents an employee with fields for id, name, and an address of type Address. The constructor initializes these fields. The display method prints the employee's details along with their address.

Significance of Aggregation

Independence: The Address objects (address1 and address2) are created independently of the Emp objects. They can exist without the Emp objects.

Reusability: An Address object can be reused across multiple Emp objects if needed, demonstrating the flexibility of aggregation.

HAS-A Relationship: The Emp class has a reference to the Address class, clearly showing that an employee "HAS-A" address.

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.

Advantage of method overloading - Method overloading increases the readability of the program.

There are two ways to overload the method in java - By changing number of arguments & By changing the data type

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```java
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args)
{System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

```
main with String[]
```

Type Promotion - When method overloading involves type promotion, Java promotes one data type to another if a matching method signature is not found. Here's a detailed explanation with examples provided.

Example 1: Method Overloading with Type Promotion

```java
class OverloadingCalculation1 {
    void sum(int a, long b) {
        System.out.println(a + b);
    }

    void sum(int a, int b, int c) {
        System.out.println(a + b + c);
    }

    public static void main(String args[]) {
        OverloadingCalculation1 obj = new OverloadingCalculation1
        obj.sum(20, 20); // second int literal will be promoted t
        obj.sum(20, 20, 20);
    }
}
```

Example 2: Method Overloading with Matching Type Arguments

```java
class OverloadingCalculation2 {
    void sum(int a, int b) {
        System.out.println("int arg method invoked");
    }

    void sum(long a, long b) {
        System.out.println("long arg method invoked");
    }

    public static void main(String args[]) {
        OverloadingCalculation2 obj = new OverloadingCalculation2
        obj.sum(20, 20); // int arg sum() method gets invoked
    }
}
```

Behavior: EXAMPL 1

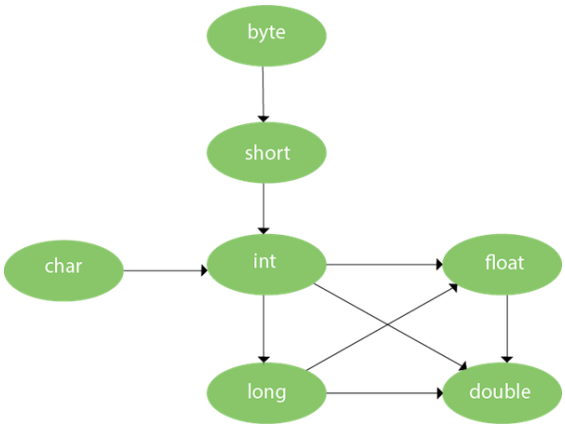
When obj.sum(20, 20); is called, there is no exact match for the method signature. Java promotes the second int argument to long, matching the method void sum(int a, long b). The output is 40.

When obj.sum(20, 20, 20); is called, it exactly matches void sum(int a, int b, int c). The output is 60.

Behavior: EXAMPLE 2

When obj.sum(20, 20); is called, the method signature void sum(int a, int b) exactly matches the arguments. Therefore, there is no need for type promotion, and the method void sum(int a, int b) is invoked. The output is int arg method invoked.

Type promotion happens based on this image -



Example: Method Overloading with Type Promotion in Case of Ambiguity

java Copy code

```java
class OverloadingCalculation3 {
    void sum(int a, long b) {
        System.out.println("a method invoked");
    }

    void sum(long a, int b) {
        System.out.println("b method invoked");
    }

    public static void main(String args[]) {
        OverloadingCalculation3 obj = new OverloadingCalculation3
        obj.sum(20, 20); // now ambiguity
    }
}
```

Behavior: AMBIGUITY

The method call obj.sum(20, 20); involves two integer arguments. Neither sum(int a, long b) nor sum(long a, int b) exactly matches the call.

Java will attempt to promote the int arguments to match the method signatures.

Promotion options:

First argument (int 20) can be promoted to long, and the second argument (int 20) remains int, matching sum(long a, int b).

Second argument (int 20) can be promoted to long, and the first argument (int 20) remains int, matching sum(int a, long b).

Because both methods can be matched through promotion, the compiler cannot decide which method to call, resulting in an ambiguity. Hence Compile Time Error

To resolve this ambiguity, you can cast one of the arguments explicitly so that the method call matches a specific method signature. -> obj.sum(20, (long) 20);

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

Method overriding is used to provide the specific implementation of a method which is already provided by its superclass. Method overriding is used for runtime polymorphism

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.

- There must be an IS-A relationship (inheritance).

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.

```
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
```

```
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInte
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass. This feature allows the overridden method in the subclass to return a subtype of the return type declared in the superclass. Covariant return types provide more flexibility and enhance type safety. In the above example, getSound() in Dog and Cat returns the specific type Dog and Cat, respectively. This enables calling methods specific to those subclasses without additional casting.

```java
class Animal {
    Animal getSound() {
        return this;
    }
}

class Dog extends Animal {
    @Override
    Dog getSound() {
        return this;
    }

    void bark() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    @Override
    Cat getSound() {
        return this;
    }

    void meow() {
        System.out.println("Meow!");
    }
}
```

```java
public class CovariantReturnTypeExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.getSound().bark();  // Output: Woof!

        Cat cat = new Cat();
        cat.getSound().meow();  // Output: Meow!
    }
}
```

1) Covariant return type assists to stay away from the confusing type casts in the class hierarchy and makes the code more usable, readable, and maintainable.
2) In the method overriding, the covariant return type provides the liberty to have more to the point return types.
3) Covariant return type helps in preventing the run-time *ClassCastExceptions* on returns.

Take this for example -

```java
class A1 {
    A1 foo() {
        return this;
    }

    void print() {
        System.out.println("Inside the class A1");
    }
}

// A2 is the child class of A1
class A2 extends A1 {
    @Override
    A1 foo() {
        return this;
    }

    @Override
    void print() {
        System.out.println("Inside the class A2");
    }
}
```

```java
// A3 is the child class of A2
class A3 extends A2 {
    @Override
    A1 foo() {
        return this;
    }

    @Override
    void print() {
        System.out.println("Inside the class A3");
    }
}

public class CovariantExample {
    // main method
    public static void main(String[] args) {
        A1 a1 = new A1();
        a1.foo().print(); // Output: Inside the class A1

        A2 a2 = new A2();
        a2.foo().print(); // Output: Inside the class A2

        A3 a3 = new A3();
        a3.foo().print(); // Output: Inside the class A3
    }
}
```

In the above program, class A3 inherits class A2, and class A2 inherits class A1. Thus, A1 is the parent of classes A2 and A3. Hence, any object of classes A2 and A3 is also of type A1. As the return type of the method foo() is the same in every class, we do not know the exact type of object the method is actually returning. We can only deduce that returned object will be of type A1, which is the most generic class. We can not say for sure that returned object will be of A2 or A3. It is where we need to do the typecasting to find out the specific type of object returned from the method foo(). It not only makes the code verbose; it also requires precision from the programmer to ensure that typecasting is done properly; otherwise, there are fair chances of getting the ClassCastException. To exacerbate it, think of a situation where the hierarchical structure goes down to 10 - 15 classes or even more, and in each class, the method foo() has the same return type. That is enough to give a nightmare to the reader and writer of the code.

```
class A2 extends A1 {
    @Override
    A2 foo() {
        return this;
    }

    @Override
    void print() {
        System.out.println("Inside the class A2");
    }
}
```

```
class A3 extends A2 {
    @Override
    A3 foo() {
        return this;
    }

    @Override
    void print() {
        System.out.println("Inside the class A3");
    }
}
```

In such a case - there is no confusion about knowing the type of object getting returned from the method foo(). Also, even if we write the code for the 10 - 15 classes, there would be no confusion regarding the return types of the methods. All this is possible because of the covariant return type.

////////////// Some bull shit on last comment - https://chatgpt.com/share/c9d64995-0b66-4627-861a-e9c2db99fa33

Super Keyword in Java

The super keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal {
    String color = "white";
}
```

```
class Dog extends Animal {
    String color = "black";
    void printColor() {
        System.out.println(color); // Prints the color of Dog class
        System.out.println(super.color); // Prints the color of Animal class
    }
}
```

super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}
class Dog extends Animal {
    void eat() {
        System.out.println("eating bread...");
    }
    void bark() {
        System.out.println("barking...");
    }
    void work() {
        super.eat(); // Calls the eat() method from Animal class
        bark();      // Calls the bark() method from Dog class
    }
}
```

```
class TestSuper2 {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.work();
    }
}
```

super is used to invoke parent class constructor.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
```

```
animal is created
dog is created
```

c

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement. So if we remove supere(); from Doge() function then also animal is created is printed, because compiler provides default Constructor

We can use supere() in case of constructor chaining in real world examples, so if there is a Person class with person constructor Person(int id,String name){ this.id=id; this.name=name; }, then super can be used when creating obj employee after Emp extends Person -> Emp(int id,String name,float salary){ super(id, name); //reusing parent cOnstructor and this.salary = salary; }

Instance initializer block

Diamond Problem –

The diamond problem is a classic issue that arises in multiple inheritance scenarios, particularly in object-oriented programming languages that allow a class to inherit from more than one parent class. This problem is called the diamond problem because the inheritance structure resembles a diamond shape when drawn out.

Here is a detailed explanation of the diamond problem in the context of Java:

Multiple Inheritance and the Diamond Problem

In multiple inheritance, a class can inherit behaviors and attributes from more than one parent class. This can lead to ambiguity if the parent classes have methods or fields with the same signature. The diamond problem is a specific case of this ambiguity.

Diamond Problem Scenario

Consider four classes: `A`, `B`, `C`, and `D`. The class `B` and class `C` both inherit from class `A`, and class `D` inherits from both `B` and `C`. The structure looks like this:

```
  A
 / \
B   C
 \ /
  D
```

If class `A` has a method `foo()`, and both `B` and `C` inherit this method, class `D` will have two inherited implementations of `foo()`—one from `B` and one from `C`. This creates ambiguity: which `foo()` should `D` use?

Java and Multiple Inheritance

Java avoids the diamond problem with classes by not allowing multiple inheritance (a class cannot extend more than one class). However, Java allows multiple inheritance of interfaces, which can still lead to the diamond problem. Here's an example using interfaces:

```java
interface A {
    void foo();
}

interface B extends A {
    @Override
    default void foo() {
        System.out.println("B's implementation of foo");
    }
}

interface C extends A {
    @Override
    default void foo() {
        System.out.println("C's implementation of foo");
    }
}

class D implements B, C {
    // D must resolve the conflict between B and C
    @Override
    public void foo() {
        // Explicitly choose which foo() to call or provide a new implementation
        B.super.foo();
        // or
        // C.super.foo();
        // or
        // custom implementation
        System.out.println("D's own implementation of foo");
    }
}

public class Main {
    public static void main(String[] args) {
        D d = new D();
        d.foo();  // This will call D's implementation of foo
    }
}
```

Explanation

1. **Interface `A`**: Declares a method `foo()`.

2. **Interface `B`**: Extends `A` and provides a default implementation of `foo()`.

3. **Interface `C`**: Extends `A` and provides a different default implementation of `foo()`.

4. **Class `D`**: Implements both `B` and `C`.

When `D` implements both `B` and `C`, it inherits two conflicting default implementations of `foo()`—one from `B` and one from `C`. Java forces `D` to resolve this conflict by overriding `foo()` and explicitly choosing which default method to call or providing its own implementation.

Resolving the Diamond Problem

In the example above, `D` overrides `foo()` and can:

- Call `B`'s implementation with `B.super.foo()`

- Call `C`'s implementation with `C.super.foo()`

- Provide a new implementation that may combine or modify behaviors from `B` and `C`.

This ensures that the ambiguity is resolved, and `D` has a clear and unambiguous implementation of `foo()`.

Summary

The diamond problem occurs in multiple inheritance when a class inherits from two classes that both inherit from a common superclass, leading to ambiguity in which inherited method or field to use. Java avoids this issue with classes by disallowing multiple inheritance but allows it with interfaces. To handle this in interfaces, Java requires explicit resolution by the subclass, ensuring clear and unambiguous behavior.
