1. **What is Binding and how many types of binding are there in oops?**

Ans:

Resolving function call with function body is called as Binding.

There are 2 types of bindings: early binding and late binding.

Early binding means resolving function call with function body at the compilation time. In Java early binding happens when a function is "private" or "static" or "final".

Late binding means resolving function call with function body at the runtime.

For the methods other than "private" or "static" or "final" we have always late binding in java.

1. Explain Java access specifiers.

private:-

Accessible in the same class only.

public:-

Accessible everywhere.

<default> :-

Accessible everywhere in the same package only.

protected:

Apart from everywhere in the same package, protected member can be accessible in the child class of different package.

1. **What is the difference between constructor invocation in case of is-a and has-a relationship?**

Ans:

suppose we have following hierarchy:

class A { // Parent class (IS-A)

A() {

System.out.println("A Constructor");

}

}

class B { // HAS-A object

B() {

System.out.println("B Constructor");

}

}

class C extends A { // C IS-A A (Inheritance)

B objB = new B(); // C HAS-A B (Composition)

C() {

System.out.println("C Constructor");

}

}

and in the main function we instantiate class "C" in the following way:

public class Test {

public static void main(String[] args) {

C obj = new C(); // Creating object of C

}

}

output would be:

A Constructor // (1) IS-A: Parent class constructor runs first

B Constructor // (2) HAS-A: B's constructor runs in declaration order

C Constructor // (3) Finally, C's constructor runs

So the conclusion is:

a) C obj = new C(); is executed.

b) Since C extends A, the parent constructor (A()) runs first.

c) After the parent (A) constructor completes, Java initializes all HAS-A objects in their declaration order (i.e., B objB = new B(); triggers B's constructor).

d) Finally, after all dependencies are initialized, the C constructor executes.

1. **What are the rules for overriding in java?**

Ans:

1) arguments must be same otherwise it becomes "overloading".

2) returntype of overriding can be co-variant.

3) overriding method must be having same or more accessibility as compare to overridden method.

4) overriding and checked-exception rule :

a) overriding method may not declare any checked exception.

b) overriding method can declare same checked exception or its sub-type declared by overriden method.

c) overriding method can not declare checked exception not declared by overriden method.

1. **What is Co-variant return type in case of overriding?**

Ans:

if overridden method has "parent" as a return type then overriding method can have "child" as a return type. E.g. if overridden method has a return type “java.lang.Object” then overriding method can have “java.lang.String” or any of the child class of “Object” class as a return type.

1. **What is the meaning of upcasting in Java?**

Ans:

Upcasting in java is, to assign child class object to parent class reference.

1. **What is the role played by Compiler in case of late binding? Explain with the help of example.**

Ans:

Suppose we have,

base ref=new sub1(); // upcasting

ref.disp(); // late binding

when compiler encounters "ref.disp()" ,

for the compiler "ref" is of type "base". So compiler searches "disp()" inside "base".

if it's there,accessible and non-final,non-static compiler writes an instruction for runtime to follow. That instruction is

" during runtime check the content of "ref" and invoke "disp()" accordingly.

1. **What is downcasting in java and why do we need it?**

Ans:

Downcasting in Java is **the process of converting a superclass reference into a subclass reference**. It is typically done when an object is upcasted earlier, and we need access to subclass-specific methods.

**Syntax of Downcasting**

Subclass obj = (Subclass) superClassReference;

**Explicit casting ((Subclass)) is required** because Java does not allow implicit downcasting.

1. **What is the risk involved in downcasting in java and what is the solution for that?**

Ans:-

**Risk Involved in Downcasting in Java**

The **biggest risk in downcasting** is that if we cast an object **incorrectly** (i.e., the object is not actually an instance of the subclass), Java will throw a **ClassCastException at runtime**.

**Example of Incorrect Downcasting (Risk)**

class Animal {

void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

void meow() {

System.out.println("Cat meows");

}

}

public class Test {

public static void main(String[] args) {

Animal animal = new Cat(); // Upcasting a Cat to Animal

Dog dog = (Dog) animal; // Incorrect Downcasting!

dog.bark(); // Runtime error: ClassCastException

}

}

**Output:**

Exception in thread "main" java.lang.ClassCastException: class Cat cannot be cast to class Dog

**Problem:** animal is actually a Cat, but we tried to cast it to Dog. This results in **ClassCastException at runtime**.

**Solution: Use instanceof Before Downcasting**

To **avoid ClassCastException**, we should **always check the actual type** of the object before downcasting using the instanceof operator.

**Safe Downcasting with instanceof**

if (animal instanceof Dog) { // Checking before downcasting

Dog dog = (Dog) animal;

dog.bark();

} else {

System.out.println("Downcasting not possible!");

}

**Output (Safe Execution)**

Downcasting not possible!

**This way, we prevent ClassCastException by ensuring that downcasting happens only if the object is truly of the expected type.**

1. **What does instanceof operator check?**

Ans:

The instanceof operator in Java is used to **check (during runtime) whether the content of a given reference ( i.e. to where it is referring ) is an instance of a specific class or its subclass**.

e.g.

ref instanceof Sample

Returns true if the content of “ref” is an instance of Sample class (or its subclass).

Returns false if the content of “ref” is not an instance of Sample class (or its subclass).

1. **What is polymorphism in java? explain it with the real life examples (at least 2 examples)**

Ans:

Polymorphism means object has many forms.

Actually there are two types of polymorphism : Compile-time and Run-time polymorphism.

Compile-time polymorphism means resolving function call with function body at compile time. Runtime polymorphism means resolving function call with function body during runtime.

**compile-time polymorphism happens only in private, static, and final methods.**

✔ **In Java, true compile-time resolution happens only for private, static, and final methods.**  
✔ **Method overloading is resolved based on reference type but still follows runtime method resolution in some cases.**  
✔ **All non-static, non-final, and non-private methods are resolved at runtime due to dynamic method dispatch.**

1. **What is abstract class in java?**

Ans:

in java as soon as u define a class with "abstract" keyword, class becomes abstract.

abstract class cannot be instantiated. It is because abstract class is incomplete class ( it has got at least one contract i.e. abstract method )

abstract class can contain abstract as well as non-abstract methods.

abstract method is a method which is declared with "abstract" keyword. ( it cannot be private)

a child class of an abstract class has to provide implementation to the method which is declared "abstract" in parent class or else make child class also "abstract".

1. **Can a class be “abstract” and “final” both?**

Ans:

A class can not be "abstract" and "final" both. It is because "abstract" encourages inheritance and "final" stops inheritance.

1. **Why do we need a constructor in the abstract class as abstract class cannot be instantiated at all?**

Ans:

Yes we cannot instantiate abstract class but when we instantiate child class of that abstract class, child class constructor needs to invoke parent class constructor. For this purpose we need constructor in the abstract class.

1. **Can an Abstract Class Have a final Method in Java?**

Ans:

Yes, abstract class can contain final method. Obviously this cannot be overridden by child class but can be easily inherited.

1. **Why do you upcast?**

Ans:

Upcasting allows us to write methods that work with any subclass of a parent class without being tied to a specific subclass. That is using parent class reference only we can invoke method of any child class polymorphically. It is basically a "program to interface" concept which always gives us maintenance advantage even if there is a change in the inheritance hierarchy.

1. **Why do you downcast?**

Ans:

suppose we have written a function which has an argument of "base type" so that we can pass any child object i.e. we follow "program to interface" concept. In this scenario, if you want to invoke a particular child class method [which is not available in parent class] we need to downcast parent class reference to that particular child class reference.

1. **What are the scenarios in which we cannot instantiate the class?**

Ans:

1. When the class is abstract

Or

1. When the class has private constructor.
2. **If at all you want to ensure that people should not create the instance of a class which approach would you follow? making class abstract or making a constructor private? and why?**

Ans:

If we want to ensure that class shouldn’t be instantiated at all, we should make it abstract instead of making its constructor private. The advantage of making class abstract instead of making it constructor private is, though the class cannot be instantiated but we can always inherit that class. Private constructor stops inheritance.

1. **Why can’t child class reference refer to parent class object? Explain with the help of an example.**

Ans:

Consider the following example:

class base

{

void disp()

{

}

}

class sub extends base

{

void print()

{

}

}

public class Demo

{

p.s.v.main()

{

base ref=new base();

sub ref2=(sub)ref; // will fail but why?

discuss

}

}

let's assume that

sub ref2=(sub)ref; works at runtime

now what if we say:

ref2.print();

when compiler had encountered this statement compiler must have written late binding instructions. As per those instructions since the content of "ref2" during runtime is "base" and base does not have "print()" method , it will be a problem. We shouldn't get this problem that's why JVM does not allow us to make "sub class reference" refer to "base class".