Regarding **Up casting**

The most important aspect of inheritance is the relationship expressed between the new derived class and the base class. This relationship can be summarized by saying,

The new class "IS A" type of the existing class.

eg : Student is of Person type or Faculty is of Person type.

Person p=new Student(....);

System.out.println(p);//println – is implicitly virtual method of PrintStream class. JVM resolves it by the type of the object , i.e. it will call toString of Student class.

Here we are treating a subclass object(Student) as its **supertype** (superclass).
Thanks to inheritance, this conversion (climbing up the hierarchy) happens **implicitly** in Java (no cast needed).

- You can't access subclass-specific methods directly
- You can gain abstraction i.e. → work with common super class reference
- It enables **runtime polymorphism**.

As another example, consider a base class called Fruit that represents any fruit, and a derived class called Mango. Because inheritance means that all of the methods in the base class are also available in the derived class, any message you can send to the base class can also be sent to the derived class. If the Fruit class has a taste( ) method, so will Mango.

This means we can accurately say that a Mango object is also a type of Fruit.

eg . Fruit anyFruit=new Mango(...);

When is down casting required ?

In the **indirect referencing**(super class reference --> sub class object ,

eg - Person p =new Student(....);)

when you are invoking sub class specific functionality.

eg – System.out.println(p);//p.toString() : down casting is not required since Person already has toString . JVM uses run time polymorphism and - calls Student's toString.

But , if you try to invoke Student specific functionality –

p.submitFeedback(…);

It will result into **javac error** , since submitFeedback() method is not found in Person class.

Solution -

Use explicit cast - down casting (equivalent to narrowing conversion, from generalization → specialization , climbing down the hierarchy)

e.g. **((Student)p).submitFeedback();**//no error , calls Student's submitFeedback()

Check the code below , carefully .

p=new Faculty(....);//up casting

System.out.println(p);//no erros ! , JVM - invokes Faculty's toString , using run time polymorphism

((Student)p).feedback();//run time error - java.lang.ClassCastException : since

Faculty cannot be cast into a Student .

Solution

1. use **instanceof** keyword

OR

2. use **Java Reflection API (getClass method)**

instanceof

- keyword in java

- used for testing run time type information.(RTTI)

- It is a binary operator

The **JVM keeps metadata** about objects(in method area), so even when you upcast an object to a superclass or interface, the JVM still knows the **actual type** of the object at runtime.

This is what makes **polymorphism**, instanceof, and **downcasting safety** possible.

“instanceof” operatort checks whether a reference variable is pointing to an object of a particular type (class, subclass, or interface).

The instanceof in java is also known as type **comparison operator(in run time)** because it compares the instance with type. It returns either true or false. It allows you to check whether an object is an instance of a particular class, subclass, or interface.

Safe Casting: It's often used before down casting a reference   to a specific type to prevent ClassCastException run time error.

Syntax:

objectRef instanceof ClassOrInterface

It returns -

true: If the reference variable pointing to an object , is an instance of the specified Class/Interface or any of its subclasses or implementations.

false: If the object is not an instance of the specified Class/Interface, or if the object is null.

- If a class extends another class, an object of the subclass is also considered an instance of the superclass.

-It can also check if an object implements a specific interface.

- null Checks: instanceof returns false when the object being checked is null.

Best Practices:

Use instanceof to ensure type safety before casting.

Consider below inheritance hierarchy

eg ---Object <----Emp <---Mgr <---SalesMgr

Object <---- Emp <--- Worker

What will it return ?

Emp e =new Mgr(...);

e instanceof Mgr - true

e instanceof Emp - true

e instanceof Object -true

e instanceof SalesMgr   -false

e instanceof Worker -false

e=null;

e instanceof Emp/Mgr/SalesMgr/Worker/Object   -false