(read oops notes)

* Inheritance ⇒ (when one class access property of another class)

When we ~~construct~~ create a new class from existing class in such a way that the new class access all the ~~function~~ features and properties of existing class called as Inheritance.

Note →

i) In java extends keyword is used to perform inheritance

ii) It provides code reusability

iii) We can't access private members of class through inheritance

iv) Sub class contains all the features of super class so we should create the object of sub class.

v) Method overriding only possible through inheritance.

Syntax →

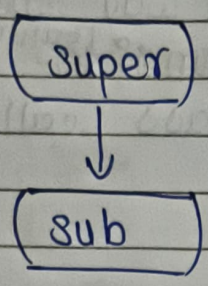
```
Class A          (super class)
{
}
class B extends A (sub class)
{
}
```

features of super class

when we use inheritance we have to create object of sub class because subclass already inherited

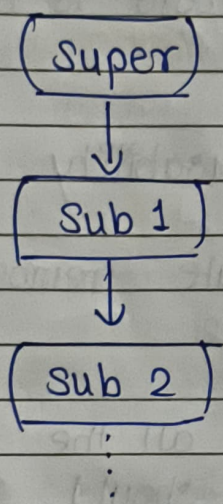
Types ⇒

① Single / Simple inheritance :→



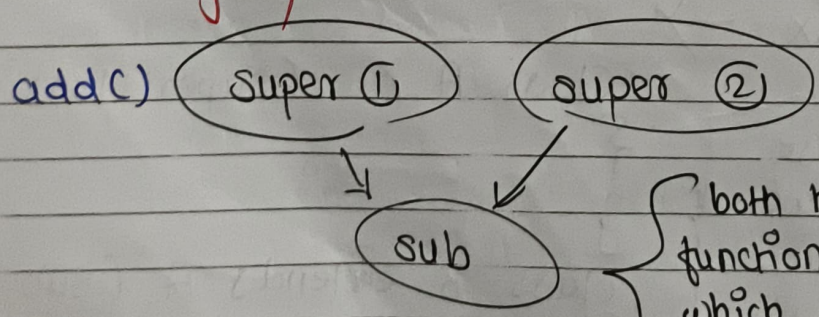
(sub class access the features of super class)

② Multi-level inheritance :→



(sub1 class access features of super class
sub 2 can access features of both sub1 class & super class)

Multiple Inheritance does not supported by Java to avoid confusion caused by diamond problem, where same method from different parent class creates ambiguity

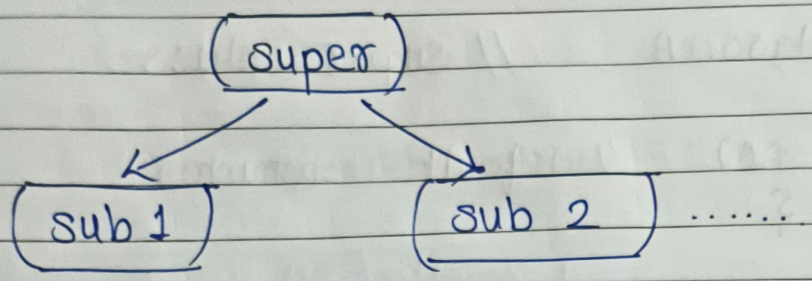


{ both having same function i.e add() which creates ambiguity }

④ Hybrid Inheritance ⇒

{also not supported in java} combination of simple, multi-level & Hierarchical (two or more types of Inheritance)

③ Hierarchical Inheritance ⇒



* Super keyword ⇒

Super keyword refer to the object of super class, it is used when we want to call the super class variable, method & constructor through sub class object.

Note →

i) whenever the super class and sub class variable and method name both are same than it can be used only.

ii) To avoid the confusion between super class and sub classes variables and methods that have same name we should use super keyword.

3 ways to implement super keyword

i) variable
ii) method
iii) constructor.

Syntax →

```
class A          // super class
{
    A()    // default constructor
    {

    }
}
class B extends A          // sub class
{
    B()    // default constructor
    {
      //super();   // by default place is
                     reserved in constructor
    }
}
```

Example →

① variable →

```
class A
{
    int a=10;
}
class B extends A
{
    int a=20;

    void show()
    {
      system.out.println(a);
    }system.out.println(super.a);
}
```

```
Class Test
{
    public static void main (String[] args)
    {
        B ref = new B();

        ref.show();
    }
}
```

Output →
20
10

② Method →

```
Class A
{
    void show()
    {
        System.out.println ("Hello");
    }
}

Class B extends A
{
    void show()
    {   super.show();
        System.out.println ("Welcome");
    }
}

Class Test
{ psvm
    {
        B ref = new B();
        ref.show();
    }
}
```

output →
Hello
welcome

③ Constructor →

① Class A

```
{
    A()
    {
        System.out.println("Hello");
    }
}
Class B extends A
{
    B()
    {
        // super();
        System.out.println("welcome");
    }
}
Class Test
{
    public static void main(String[] args)
    {
        B ref = new B();
        ~~ref~~..
    }
}
```

Output →
Hello
welcome

{ No need to write super keyword by default compiler can add }

But if in class A if constructor A is parameterized then it is compulsory to write super keyword in sub class constructor

(11)

```
class A
{
    A (int a)
    {
      System.out.println ("Hello"+ a);
    }
}
class B extends A
{
    B ()
    {
      Super(10);
      System.out.println ("World");
    }
}
class Test
{
    public static void main(String[] args)
    {
        B ref = new B();
    }
}
```

Output →

Hello 10
world

It is "is-like-a" relationship.
Example → student is not a dancer.
student is like a dancer.

Interface ⇒ (Explore Types of Interface)

Interface is just like a class, which contains only abstract method.
To achieve interface java provides a keyword called implements

Note ⇒

i) Interface methods are by default public and abstract

ii) Interface variables are by default public static final.

iii) Interface method must be overriden inside the implementing classes. (compulsory)

iv) Interface nothing but deals between client & developer (Contract based)

Example →

```
import java.util.Scanner;
interface client
{   int a = 5; // public & static & final ← (by default)
    void input(); // public & abstract ←
    void output(); // public & abstract ←
}

class Developer implements client
{
```

```
String name;
double salary;

public void input()
{
    Scanner sc = new Scanner(System.in);

    System.out.println("Enter Username:");
    name = sc.nextline();
    System.out.println("Enter Salary:");
    salary = sc.nextDouble();
}
public void output()
{
    System.out.println(name + " " + salary);
}
}
Class Main
{
    public static void main(String [] args)
    {
        client c = new Developer();
        c.input();
        c.output();
    }
}
```

With the help of interface in this case Developer does not give source code to client, instead of that he gives only setup file

Here we can create Interface refference variable but we have to create object of Class only

(Abstraction is achieved by 2 ways)

① Abstract class

② Interface

(~~Incomplete / Halfminded~~)

* Abstract Class →

A class which contains the abstract keyword in its declaration is called abstract class

Note →

i) We can't create object of abstract class but we can create refference of extends class (inherited class)

Example →

```
abstract class Animal
{
}
class Dog extends Animal
{
}
class Main
{
    public static void main(String[] args)
    {
        Animal a = new Dog();
    }
}
```

{object of abstract class}

{refference of inherited class}

ii) It may or may not contain abstract method

iii) It can have abstract and non-abstract method

iv) To use an abstract class, you have to inherit it from sub class.

v) If a class contain partial implementation then. We should declare a class as abstrat (i.e in class if function has no defination)

Abstract method ⟹

A method which contain abstract modifier at the time of declaration is called abstract method.

Note →

i) It can only be used in abstract class

ii) It doesn't contain any body "{ }" and always ends with " ; ".

iii) Abstract method must be ~~executed~~ overriden in subclasses otherwise that subclass also become abstract class

iv) Whenever the action is common but implementation are different then we should use abstract method

Example →

i)
```
abstract class fruits
{
abstract void taste();
}
```

ii)
```
abstract class vehicle
{
abstract void wheeler();
}
```

Code →

```
abstract class Programming
{
  public abstract void Developer();
}
class HTML extends programming
{
  @Override
  public void Developer()
  {
    system.out.println("Tim Berners Lee");
  }
}
```

```
class Java extends Programming
{
    @override
    public void Developer(){
        System.out.println("James Gosling");
    }
}
class Main
{
    public static void Main(String[] ar)
    {
        HTML h = new HTML();
        h.Developer();
        Java j = new Java();
        j.Developer();
    }
}
```

OR

```
public static void Main(String[] args)
{
    Programming h = new HTML();
    h.Developer();
    Programming j = new Java();
    j.Developer();
}
}
```

Here we can create object refference of abstract class but we cannot create its object

* Typecasting ⇒

Converting one datatype to another datatype is called Typecasting

Two types

i) Implicit ii) Explicit

① Implicit Typecasting → (Widening Casting)

It is automatically performed by the Compiler.
Converting a smaller type to a larger type size

byte → short → char → int → long → float → double

Example →

```
Class A
{
    public static void main (String[] args)
    {
        int a=10;        // 4 Bytes
        double b;        // 8 Bytes

        b=a;             // Implicit typecasting

        System.out.println(b);
    }
}
```

② Explicit Typecasting → (Narrow Casting)

By defaut the compiler, doesn't allow the Explicit Typecasting (Manually)

Converting a larger type to a smaller size type.

Example →

```
Class A
{
  public static void main( String[] args)
  {
      double a = 10.5;   // 8 Bytes
      int b;             // 4 Bytes

      b = (int) a;       // Explicit typecasting

      System.out.println(b);
  }
```

***

In java, the object can also be typecasting like the datatype.

Upcasting & Downcasting is the type of object typecasting.

① Upcasting ⇒

i) Upcasting is the typecasting of a child object to a parent object.

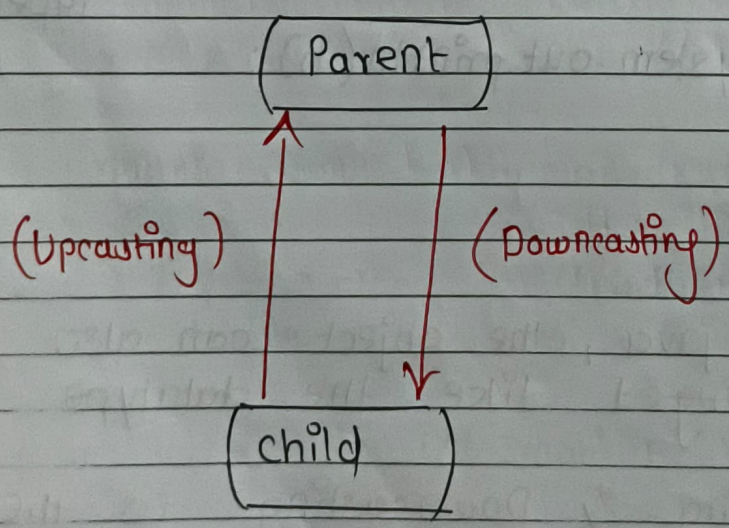
ii) Upcasting can be done implicitly it is (Automatic)

iii) With the help of upcasting we can access the overridden methods of child class

② Downcasting ⇒

i) Downcasting is the typecasting of a parent object to a child object

ii) Downcasting cannot be implicit it is (Mannual)



Example →

Let there be a parent class. There can be many children of a parent. Let's take one of the children into consideration. The children (child) inherits the properties of the parent.
Therefore, there is a "is-a" relationship between child and parent. i.e child can be implicitly upcasted to the parent. But a parent doesnot inherit the child's properties. Here we can forcefully cast a parent to a child which is known as downcasting.

Syntax of Upcasting →

```
Parent p = new child();
```

(Upcasting will be done internally & due to upcasting the object is allowed to access only parent class members and child class specified members (overridden methods) but not all the members)

Syntax of Downcasting →

```
Child c = (child) p;
```

(Downcasting has to be done externally i.e manually)

Example Code → (Upcasting)

```
Class Animal
{
    void sound() {
        System.out.println("Animal make sound")
    }
}
Class Dog extends Animal {

    void sound() {
        System.out.println("Dog barks");
    }
}
public class UpcastingExample {

    public static void main(String[] args)
    {
        Animal animal = new Dog();
                                // Upcasting

        animal.sound();
                                // calls the overridden
                                   method in Dog.
    }

}
```

Key Point →

Upcasting is implicit & safe.

Example Code → (Downcasting)

```
Class Animal
{
    void sound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {

    void sound() {
        System.out.println("Dog barks");
    }
    void fetch() {
        System.out.println("Dog fetch the ball")
    }
}
public class DownCasting Example {

    public static void main( String[] args)
    {
        Animal animal = new Dog(); //Upcasting
        Dog dog = (Dog) animal; // Downcasting
        dog.sound(); // calls overridden method
        dog.fetch(); // calls subclass specified
                                    method.
    }
}
```

Key point →

Downcasting require explicit casting & can throw a ClassCastException if not done correctly.

* (Using instanceof in downcasting helps ensure safe type conversions & avoid ClassCastException)

(No instanceof is needed for upcasting because it's always safe)

* Downcasting Example with Instanceof

```
Class Animal {

    void sound() {
        System.out.println("Animal make sound");
    }
}
Class Dog extends Animal {

    void sound() {
        System.out.println("Dog barks");
    }
    void fetch() {
        System.out.println("Dog fetch the
                                ball");
    }
}

public class DowncastingExample {

    public static void main(String[] args)
    {
```

```
Animal animal = new Dog(); //Upcasting

if (animal instanceof Dog) // check if
                              'animal' is an instance of Do
{
  Dog dog = (Dog) animal; // safe downcasting
  dog.sound();            // calls overridden method
  dog.fetch();            // calls subclass specific
                             method.
}
else {

System.out.println("The object is not an
                        instance of Dog");

}
}
}
```

Note →

i) Use instance of for safe type-casting before downcasting

ii) Avoid unnecessary downcasting when polymorphism can achieve the same behaviour.