Interface — continuation

→ If a class implements from two different interfaces & if those interfaces have methods with the same name & signature, then the class should provide a common implementation for the both methods

example:

```
interface X {                    interface Y {
    void play();                     void play();
}                                }

class Parent implements X, Y {
    void play() {
        SO.P("playing");
    }
}
```

// No problem it is allowed

→ If a class implements multiple interfaces & if those interfaces have methods with same name & different signature, then we have to provide multiple implementations (body)

ie [both method overloading & method overriding]

example:

```
interface X {                    interface Y {
    void play();                     void play(int a);
}                                }

class Parent implements X, Y {
    void play() {                // overriding
    }
    void play(int a) {           // overloading
    }
}
```

→ If we have multiple interfaces having the methods with same name & same signature & different return types, then we cannot implement these interfaces to a single class at the same time

```
interface X {                    interface Y {
    int play();                      byte play();
}                                }
```

class Alpha implements X, Y {

int play() {

}

byte play() {

}

}

Common implementation is not possible

ambiguous

// show error incompatible return type for X.play()

incompatible return type for Y.play()

method call

→ obj.play()

Note:

→ So the only solution in this case implement only one interface (either X or Y) to the class.

→ We cannot have constructors inside the interface & any other blocks also not allowed

→ If a class is implementing the interfaces, if those interfaces having variables with the same name, then the class cannot access directly variable, because there is a ambiguous situation, to overcome this the variables can be accessed with the help of interface name.

i.e 'interface_name.variable

eg:

```
interface X {
    public static final int a = 10;
}

interface Y {
    int a = 20;
}

class Alpha implements X, Y {
    p.s.v.m(String args[]) {
        S.O.Pln(a) // field ambiguous
        S.O.Pln(X.a) ✓
        S.O.Pln(Y.a) ✓
    }
}
```

```
class X {
    static int a=10;
    int b;
}

new X() ✗

X.a ✓
```

[possibilities]

	C		I
→	A	implements	B

→ A (C) implements B, C, D (I) (I) (I)

→ A extends B < A, B are classes / A, B are interfaces

→ A (C) extends B (C) implements C (I)

→ A (I) extends B, C, D (I) (I) (I) — multiple inheritance ✗

Q When should we go for class (concrete), Abstract class & Interface — return (1/2)

✓ **Concrete class:** When we know the complete implementation

✗ **Abstract class:** When we know partial implementation then we can go →

m1 (); m2 () {

}

✗ **Interface:** When we have no idea about implementation & zero knowledge then we will go with interface

define ——→ interface → X, Y, Z { m1, m2, m3 }

↓ complete picture

→ Driver — class D → X, Y, Z { m1 { } , m2 { } m3 { } }

→ Q2, → Q3 — S ⊢——⊣ D

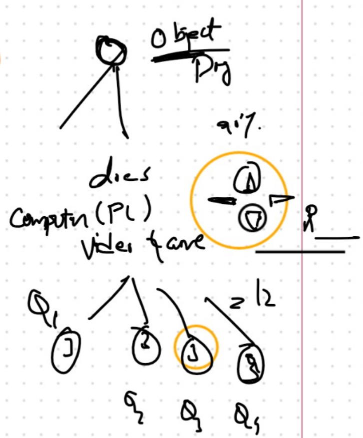
Env. , 3D models , Rendering , Audio , N-Dimensional movements

class Test extends D {
m1 () {

}
}

Can — m1

Research



Abstract class

→ In case of abstract class can have concrete methods along with abstract methods

→ We can have final methods

→ It is compulsory to declare abstract keyword for abstract methods

→ We can have instance variable

→ It can have constructor

→ We can have instance & static blocks

→ We cannot achieve multiple inheritance

Interface

→ Cannot have concrete methods

→ Cannot have final methods

→ By default it is abstract no need declare explicitly

→ We cannot have instance variables only static {by default} ↓ public static final

→ We cannot have constructor

→ We cannot have static & instance blocks

→ We can achieve multiple inheritance

Q: [Home work]

1. Need of constructor in abstract class
2. Role of constructor chaining

use case scenario

→ OOPS

4 pillars

1. Encap
2. Abstract
3. Inheritance
4. Polymorphism

Static overriding is-a has-a factory method style pattern abstract Interface

3

→ Exception handling
- → multiple catch blocks
- → final exception methods
- → creating Exception Object
- → Duckly
- → Custom Exception
- → Rethrow Exception
- → try()
- → Method handling

→ Multi tasking
→ multi threading ——↑
→ Collection
→ 1.8 functionality