"Don't ask anyone untill you yourself fail to find the ans"

— Dr Kalam

" What if ---- "

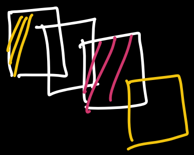
→ Write down your question

→ Write code & observe the behavior.

→ Find the why? behind the behavior.

- Google
- StackOverflow } 30 min
- Instructor / TA / Slack Grp

10-15%



Easily
Maintainable
Understand
Extensible

85-90%

Maintenance
- Testing
- Find Bug
- Fix bugs
- Refactor

Understandability
- Understand Others code
- Code review
- KT (Knowledge transfer session)

Extensibility
- Regression bugs
- Merge conflicts

- OOP (Inheritance, Abstraction, Polymorphism, encapsulation)
- SOLID
- Design Patterns
- UML

1.5 Months

* Give an object oriented design of a Birds.

Amazon
(SDE-2)

Bird
→ Properties
→ Actions / Behaviour

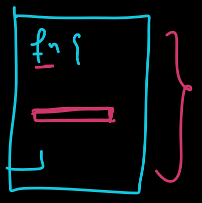
```
Bird hen = new Bird();
                  └→ Constructor*
    hen.fly();

Bird eagle = new Bird();
    eagle.fly();
```

→ Template
→ Blue Print

```
Class Bird {
  // Attribute
  string color, species;
  double wt, ht ...;
  :

  // Method
  void eat () {
     ==          → SO if I.
  }
  void fly () {
     ==
  }
  void sleep () {
     =           SO if (s
  }
  Public Bird(w, h){
       this.wt = w;
       this.ht = h;
  }
}
```

```
void fly( ) {

    if ( this.species == "hen") {
            // fly like hen.
    }
    else if ( this.species == "eagle") {
         // fly like eagle.
    }
}
```

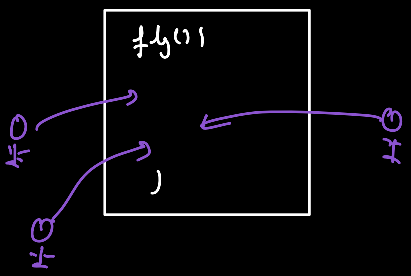
→ Single function ⇒ Multiple behaviour.

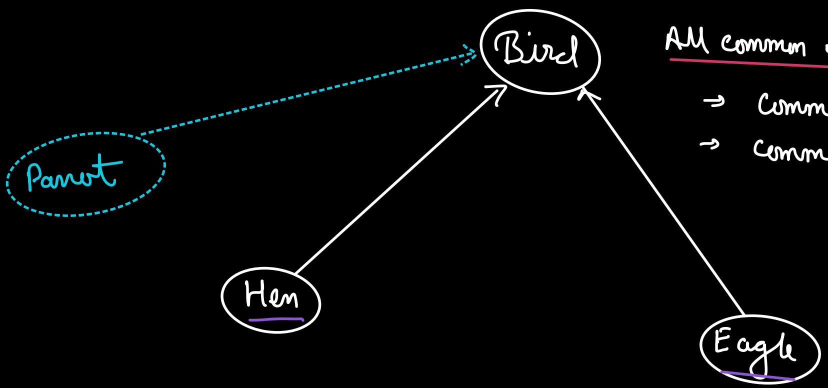


LGTM

⋮
so if/else

+! if/else





All common things
→ Common attributes wt, ht, color
→ common methods . eat, sleep
fly() {
// fly like a pigeon
}

```
class Hen extends Bird {
   // Overriding
   void fly() {
      // fly like a hen
   }
}
```

```
class Eagle extends Bird {
   // overriding
   void fly() {
      // fly like an eagle.
   }
}
```

Single Responsibility Principle → Every fn & class should have single & unique responsibility.

Open / close Principle → Write code in such a way that it is open for extension & closed for modification.

↳ Extension
↳ Modified

Q1 Should we allow anyone to create an object of Bird ?

Q2 Do we need any implementation of fly() in Bird ?

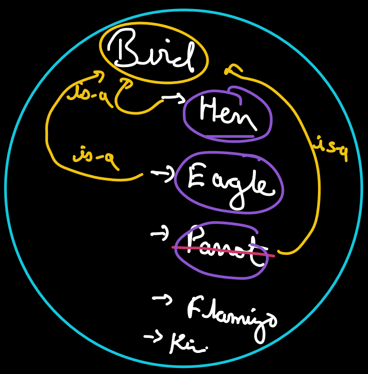
$\boxed{\text{NO}}$

abstract class → No object can be created

abstract method →
- No implementation (in parent class).
- Can only be present in abstract class
- All child classes must override it.

```
abstract class Bird {
    String color, species;      } // common Attribute
    double wt, ht

    void sleep() {
        // ===========
                                } → Common methods with common implementation
    }
    abstract void fly();
}
```

Break till 11.00PM



```
Class AngryBird {

    void render ( Hen h) {
        // Render trees
        // Render Graphics
        // Render h
        // h.fly();
    }

    void render ( Eagle e)
        // Render trees
        // Render Graphics
        // Render e
        // e.fly();

    ~~void render ( Parrot p)~~
        ~~// Render trees~~
        ~~// Render Graphics~~
        ~~// Render p~~
        ~~// p.fly();~~

}
```

Method Overloading
(Polymorphism) ⟶

```
Class AngryBird {
                          Kiwi
    void render (Bird b) {
        // render b
        // b.fly()
    }
```

Runtime Polymorphism

```
Hen h = new Hen();
abc.render ( h);
             └─is-a─→ Bird
```

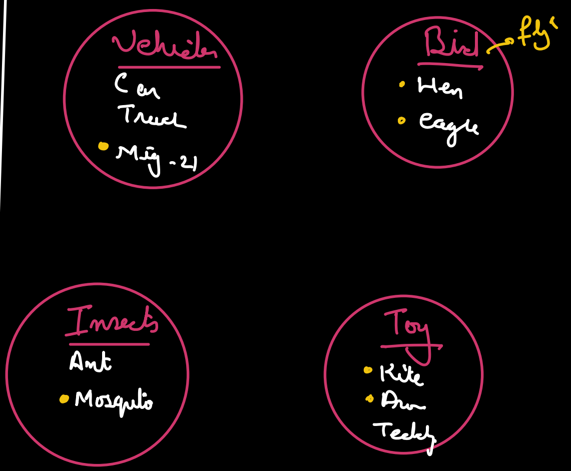
HashMap

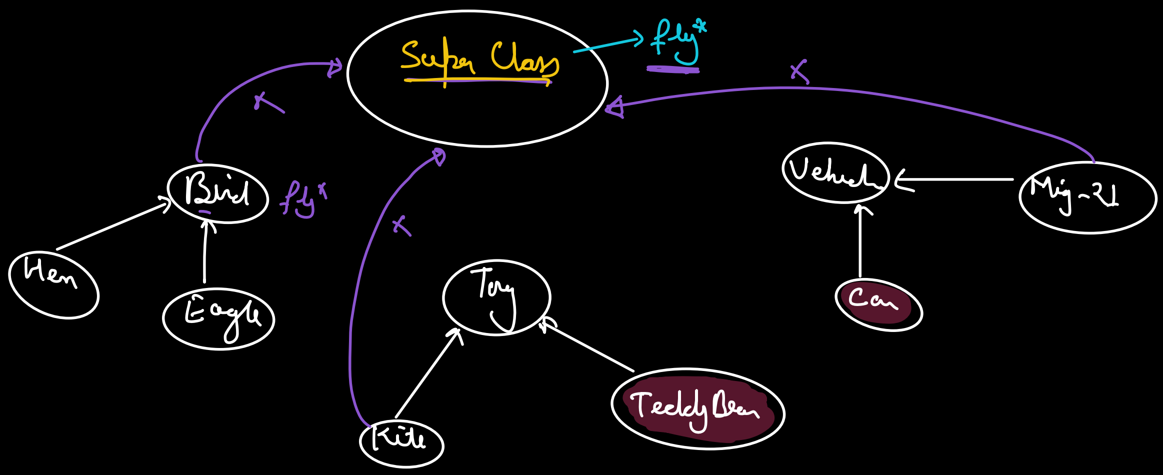
Java-7 ⟶ Java-8

All details regarding → How many birds?
→ Adding new bird?
→ Removing any bird?

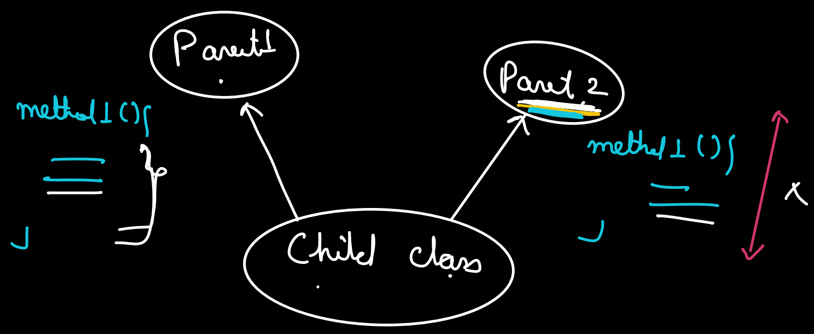
Abstraction

```
class FlyingObjects {

    void render ( flyable f ) {

        f.fly();

}
```





Diamond Inheritance Problem



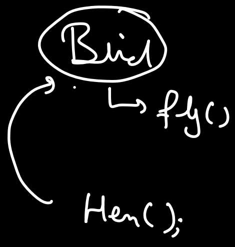
C. method1(); $\Longrightarrow$ Ambiguity

If Parent 2 has all abstract methods
└──→ Interface (Java)

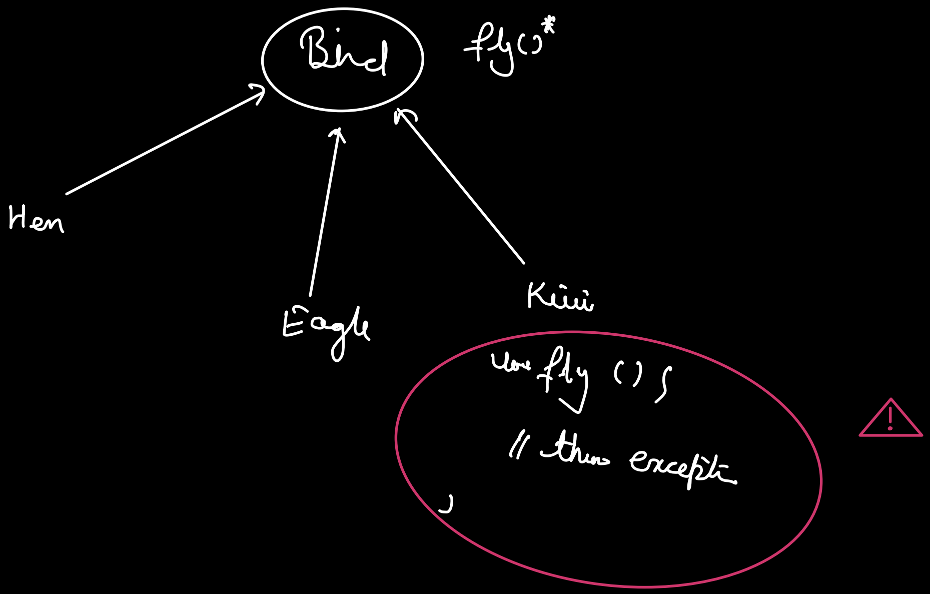
```
Interface flyable {
    void fly();
  * void flapWing();
}
```

```
class Mig-21 extends Vehicle impl flyable {
    void fly() {
        ____
    }
}
```

I → Interface Seggregation Principle



LSP → Parent class object should be replacable by child class object
- No class should be implementing methods which it is not supposed to implement.



```
void fly() {
   // throw exception
}
```

Interface $\longrightarrow$ A purely abstract class
(All methods abstract)