SOLID Principles

Advantages
- → Avoid Duplicate code
- → Easy to maintain
- → Easy to understand
- → Flexible software
- → Reduce complexity

→ is-a
→ has-a

S → Single Responsibility Principle

:- A class should have only 1 reason to change and 1 responsibility.

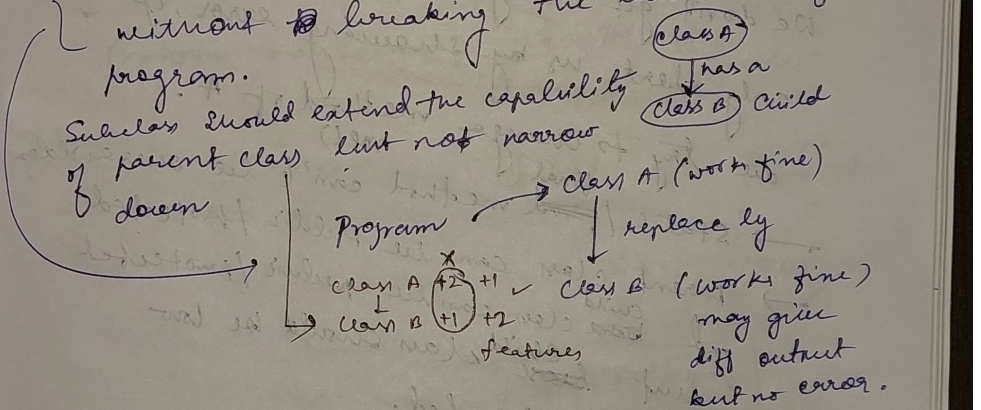
O → open / closed Principle

:- open for extension but closed for modification

L → Liskov substitution Principle

If class B is a subtype of class A, then we should be able to replace object of A with B, without breaking the behavior of program.

Subclass should extend the capability of parent class but not narrow down



I → Interface Segmented Principle

Interfaces should be such that client should not implement unnecessary function they do not need.

D → Dependency Inversion Principle

Class should depend on interface rather than Concrete Classes

Method overriding

Rules 'A' parent class and 'B' is Sub/Child class. Now say A has a method me(). B also has a same method, but different ~~bod~~ method body. So ~~wh~~ when we make a object of B, it will call the body of me() of B only, instead of me() of A.

Now to make it more robust while indicating any error, we use @Override. We don't get any added benefit. It only alerts us by showing error, if we try to change method of A.

→ static / ~~final~~ final method can't be overridden

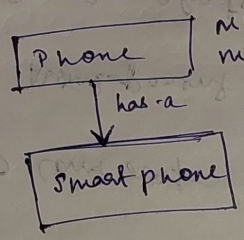
→ parent class can be public / protected and ~~child~~ child class can be public / protected but ~~base~~ child class should be low access restricted.

→ the entire function name, return type, ~~parameter~~ parameter must be same.

@Override

Abstract Class & Interface & Annotation

Dynamic method dispatch



Phone ph = new Smartphone();

- Phone → reference name
- ph → reference pointer
- new Smartphone(); → Actual object creation is run time (Polymorphism)

ph.meth1 ✓

ph.meth2 ✓ (Smartphone method)

ph.meth3 ✗ [Since we are referencing our ph as Ph]

Our actual object is new Smartphone

Smartphone smph = new Phone() ✗

abstract class and method

Abstract → Which can be created.

- Method ✓
- Static method ✗
- Obj of classes ✗
- Constructor ✗

Ex: Abstract object of class can't be created.

① A class with abstract method has to be abstract class.

② Abstract class has declaration of abstract method but the actual definition are overriden by the children of this class.

Three files + 1 main file

- Abstract Parent
- Son
- Daughter
- Abstract Main

① Constructor of a abstract class ✓

② Can not create object of abstract class

Abstract Parent abc = new Abstract Parent () ✗

③ abstract Abstract Parent ( ) ✗

we can't create abstract constructor

Constructor of a abstract class →

```
public Son(int age)
{ super(age);
}
```

```
public abstract class
Abstract Parent
{ int age;
  Abstract Parent
  public (int age)
  { this.age = age;
  }
```

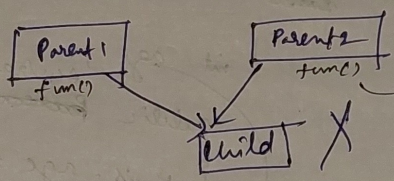
```
Son son = new Son(78);
son.age → 78
```

* Static methods can't be overridden
So Abstract static method ✗
But we can create static method in abstract class.

// final keyword
Abstract classes needs to be extended so that methods can be overridden.
If final is put before a class, it prevents from inheriting. " final abstract class Abstract parent() ✗

We can have final variables, which need to be initialized in constructor.

multiple inheritance is not supported in abstract class.



Since normal methods are allowed.
So Two different classes with same name of function will create ambiguity

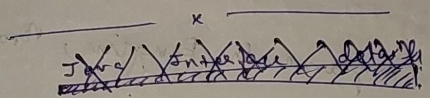
To solve this, we use

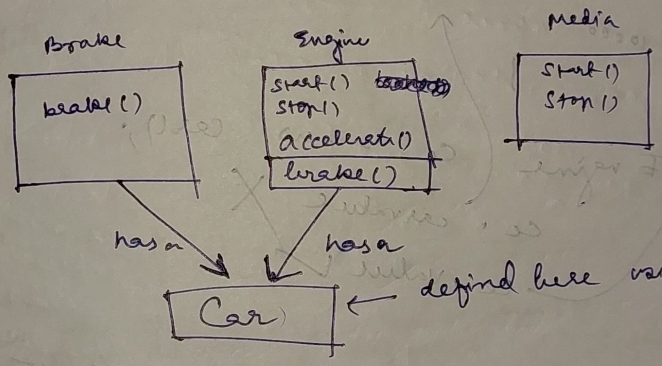
Java interfaces → contain abstract function & public
→ It is like a class but not completely
→ Abstract class with by default function as abstract and public and variable are static and final by default
→ can't create objects of Java interfaces.
→ What the class is doing, not how
→ in interface, we can't have it is doing. constructor, so how we will initialize that, is why by default final variable.

Interface → implements keyword
Abstract → extends keyword

→ public by default

class can implement more than 1 interface
but can extend only 1 abstract class.
Multiple Inheritance ~~solved~~ solved.





Note: Parent()
{
func()
}

Child extends Parent()
{
~~func~~ func()
}

Now during inheritance at compile time the func() gets mapped up the child-parent hierarchy to check compatibility / run time polymorphism.
using Interface this is not the case. Two unrelated class can implement same interface.
→ Since the functions are defined only at children level.

ClassA object = new classB();

↳ What things you access depends on this

↳ Which one of the versions you access depends on this

Suppose

in Car implements Engine, Brake

{

Engine

{ int value = 10000

}

int carvalue = 5??;

Engine ce = new car();

ce.carvalue

ce.value

// Due to this dynamic lookup method at runtime ↳ overhead

, so we should avoid Java interface in performance critical Role //

public class CDPlayer implements Media {

@ Override

start () Stop ()

{ //define } //define

}

}

public class PowerEngine implements Engine {

@ Override

start () Stop () accelerate ()

{ PowerEngine { PowerEngine { PowerEngine

} } }

public class ElectricEngine implements Engine {

@ Override

start () Stop () accelerate ()

{ Electric Engine { Electric Engine { Electric Engine

} } }

public class NiceCar {

private Media media = new CDPlayer ();

private Engine engine ;

public NiceCar () ← default constructor

{ engine = new PowerEngine ()

}

public void mediaStart()

public NiceCar (Engine engine)

{ this.engine = engine; ← parameterized constructor.

{

public void Engine start ()

{ engine.start () ← depending on initialization

}

}

Also in Nice Car

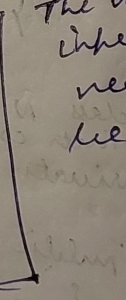
public void upgradeEngine(Engine engine)
{ this.engine = engine
}

in Main car.upgradeEngine(new PowerEngine());

Extending interfaces

class - Interface
↳ implement
interface - interface
↳ extends

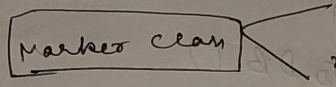
public interface A { void fun(); }
public interface B extend A { void great(); }
public class Main implements B {
@override
void fun()
{
//
}
@override
void great()
{
//
}
}

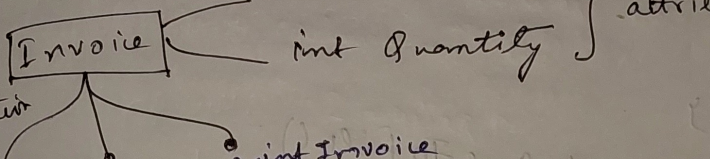


SOLID

S Single responsibility Principle

Issue :

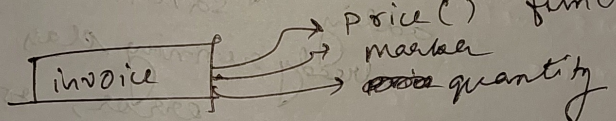


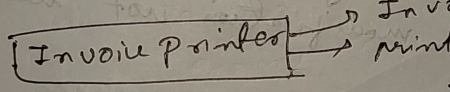


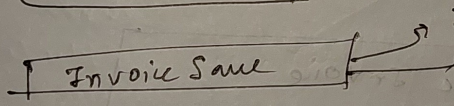
Now tomorrow, if price calculation changes then we need to change price method. Similarly if instead of SaveTo DB → Save to file

So 3 dependent responsibility

So we destructure







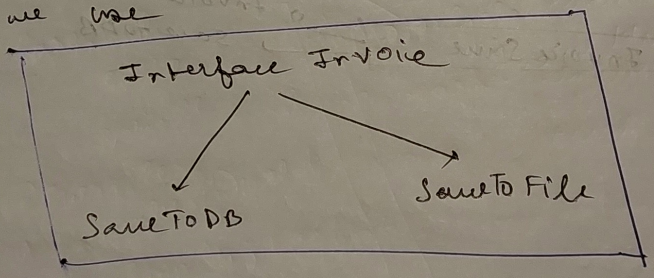
O → open for extension but closed for modification
(open/closed principle)

```
class InvoiceSave {
    public void saveToDB()
    {
    }
    public void saveToFile()
    {
    }
}
```

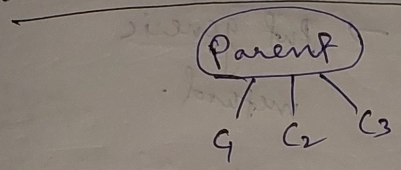
✗ not allowed

Reason: The previous class was running smoothly. Now on adding saveToFile in already running class may cause error, so modification not allowed.

So we use



L → Liskov Substitution



* boolean → true/false

* Boolean → true, false, null

Parent parobj = new C1 ()

parobj = new C2 ()

parobj = new C3 ()

The program should not break if we change the objects

ISSue

```
class Vehicle {
  int hasNumberOfWheels() {
    return 2; }
  Boolean hasEngine()
  { return true; }
```

is-a

```
MotorCycle extends
Vehicle

Bicycle Extends Vehicle
{@
  Boolean hasEngine {
  return null;
}
```

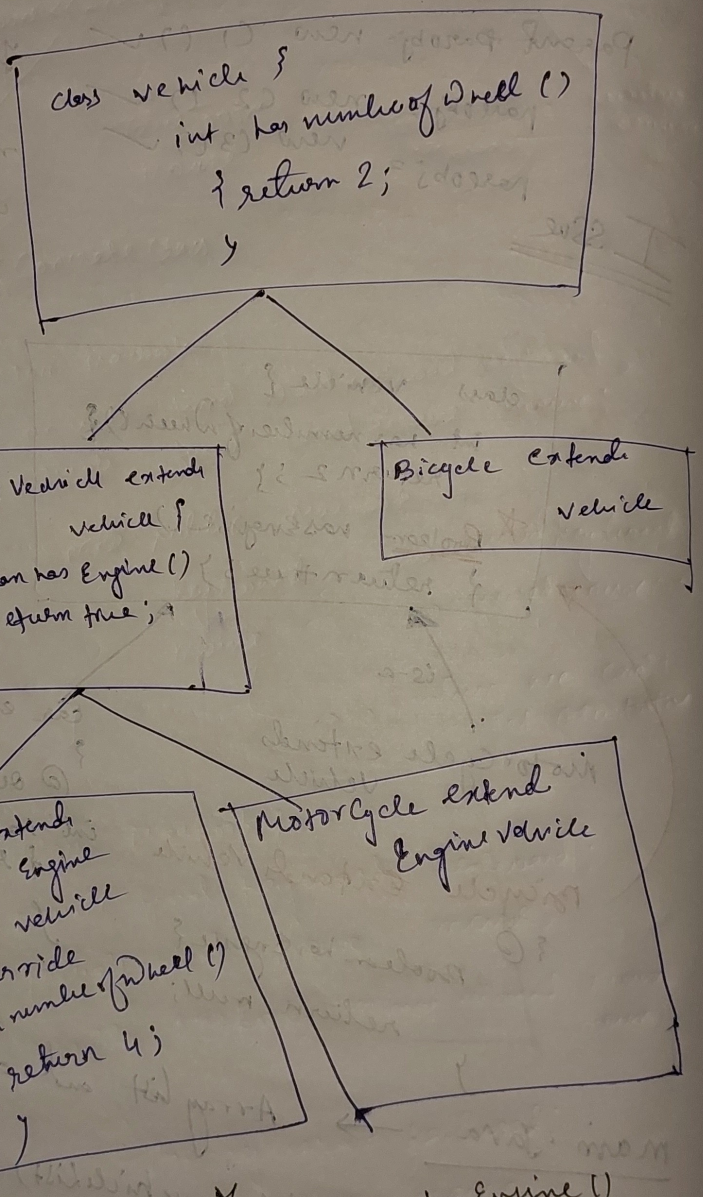
```
Car extends Vehicle
{
@Override
int hasNumberOfWheels()
{ return 4; }
}
```

main.Java → Array list and add all the 3 objects

for(Vehicle vehicle : vehicleList)

vehicle.hasEngine().toString() → Null pointer Exception

Solution In parent class → Put generic method.



Now, we can't call ✱ vehicle.hasEngine()

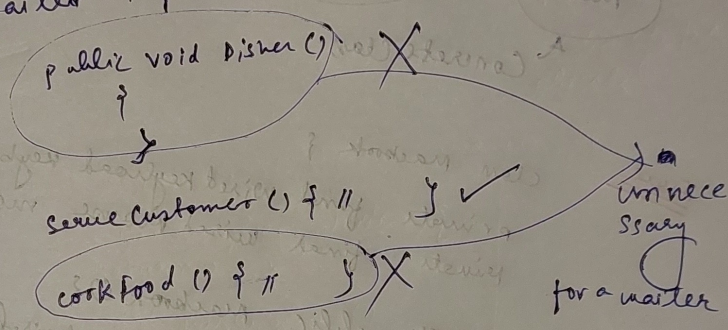
Compile Time Error → ↳ as not present in Bicycle

Interface Segmented Principle

Issue

```
interface RestaurantEmployee {
    void washDishes();
    void serveCustomer();
    void CookFood();
}
```

```
class waiter implements RestaurantEmployee {
    public void Dishes() { }
    serveCustomer() { // }
    cookFood() { // }
}
```



Solution

```
interface waiterInterface {
    void serveCustomer();
    void takeOrder();
}
```

```
interface ChefInterface {
    void cookFood();
    void decideMenu();
}
```

```
Class waiter implements waiterInterface {
    void serveCustomer()
        //
    void takeOrder()
        //
}
```