OOPS

* Use non static inside static....
  - ↳ static void fun() {
    - Classname obj = new classname
    - obj. greeting();

* initialize static :-
  - ↳ static block
  - ↳ will run only first time only...

* Singleton class
  - ↳ has only one object
  - ↳ cant have more than 1
  - ↳ if you create another obj it returns the first instance only
  - ↳ call it from other class like
    - Singleton s = singleton.getInstance();
    - " s2 = " "
    - ↳ same instance

* Final:
  - ↳ prevents the Primitive data type to change.
  - ↳ for non-Primitive, u can change the value but cant reassing it like:
    - final temp rand = new Temp();
    - rand = new Temp(...);
    - ↳ error

* Garbage collection
  - Object ob1 = new Object(a,)..
  - ob1 = new Object(b,c);
  - ↳ this points to new obj now so collecting the garbage as prev. instance...
  - ob.finalise()
  - ✓ N/2 obj can point to same reference...

* Outer class cant be static...
* if static class inside non static class then we cant create object of this class inside PSVM

* System. out. Println()
  - System ↓ class
  - out ↓ Inner class
  - Println ↓ method
  - depends on outer

```
non static A{
    non static B{
}
# A.B.method()
```

* but if inner class is static you create obj. directly, without using outer class...

```
non static A{
    static B{
        method;
}
# B.method()
```

* if Parent class at highest is box() u can still call super(); because Every class is a subclass of Object class.

* How code runs?
  - source code → Compiler (JDK) → Byte code ↓ JVM (Java virtual machine) → native code (0,1)

* non static inside static ✗
  - ↳ if something does not need an object itself how can it have something that needs objects...

* Overriding depends on objects, hence static cannot be overriden by can be inherited....

* Parent Box → l,h,w
* Child Boxweight → l,h,w,wt
  1. Box box3 = new Boxweight(1,2,3,4);
     - ↳ can't access wt
     - Box → what we can access depends on this
     - Boxweight(...) → which constructor to call only
  2. Boxweight box1 = new Box();
     - ↳ not allowed

* runtime Polymorphism / overriding :-
  - ↳ during runtime
  - ↳ same f'n and argument as Parent but body Different

* compile time PolymorPhism / overloading
  - ↳ during compilation
  - ↳ Argument type different

* Parent obj = new Child();
  - Parent ↓ what all features from that f'n you can access...
  - Child() ↓ which f'n to be called
  - ↳ thats why it cant be child obj = new Parent();

* Final Prevents overriding and Inheritence.

* Encapsulation : bundling of data and methods and restricts direct access of some of its object...
  - encal { ↳ if its variable is Public → Public methods / → direct
  - { ↳ if variable Private → getters setters to access and modify..
  - ↳ data hiding

{ Encapsulation = bundling in a single unit called class + datahiding ↓ (Private + Public method) }

* Abstraction :-
  - ↳ concept of showing important detail hiding the implementation details.

> **Real Time example:** TV Remote Button
>
> in that number format and power buttons and other buttons there.just we are seeing the butttons,we don't see the button circuits.i.e buttons circutes and wirings all are hidden.so i think its good example.

  - ↳ does not have a body.
  - ↳ another class can extend it and give it a body accordingly without caring how it internally works...
  - See v/s ↳ abstract class dont have constructor. cus if it had then we can call also but it does not have body...
  - ↳ cant have final abstract class....
  - ↳ abstract class can have normal methods..

* Interface
  - ↳ abstract class that Provides multiple inheritence
  - ↳ no such thing as immediate above class
  - ↳ No hierarchy
  - ↳ class cannot extend interface
  - ↳ Interface can extend interface
  - ↳ static method in interface must have body → bcz it cannot be overriden....

* abstract class → extends concept hence hierarchial
* Interface ↳ no hierarchy...

SOLID

1. S - Single Responsibility Principle (SRP)
   - **Definition:** A class should have only one reason to change, meaning it should have only one job or responsibility.
   - **Explanation:** Each class should focus on a single task or responsibility. If a class has multiple responsibilities, it becomes coupled, and a change to one responsibility might affect the others.

```java
// Bad Example
class User {
    void saveToDatabase() { /* code to save user to DB */ }
    void generateReport() { /* code to generate user report */ }
}

// Good Example
class UserRepository {
    void saveToDatabase(User user) { /* code to save user to DB */ }
}

class UserReport {
    void generateReport(User user) { /* code to generate user report */ }
}
```

2. O - Open/Closed Principle (OCP)
   - **Definition:** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
   - **Explanation:** You should be able to add new functionality to a class without changing its existing code. This can be achieved through inheritance and interfaces.

```java
class Rectangle {
    double length;
    double width;
}

class AreaCalculator {
    double calculate(Rectangle rectangle) {
        return rectangle.length * rectangle.width;
    }
}

// Good Example
interface Shape {
    double calculateArea();
}

class Rectangle implements Shape {
    double length;
    double width;

    @Override
    public double calculateArea() {
        return length * width;
    }
```

→ these then you have to create for each shape

] Define area on your own

3. L - Liskov Substitution Principle (LSP)
   - **Definition:** Subtypes must be substitutable for their base types without altering the correctness of the program.
   - **Explanation:** Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

```java
class Bird {
    void fly() { /* code to fly */ }
}

class Ostrich extends Bird {
    @Override
    void fly() {
        throw new UnsupportedOperationException("Ostriches can't fly");
    }
}

// Good Example
abstract class Bird {
    abstract void move();
}

class FlyingBird extends Bird {
    @Override
    void move() {
        fly();
    }

    void fly() { /* code to fly */ }
}
```

Like (I)

4. I - Interface Segregation Principle (ISP)
   - **Definition:** Clients should not be forced to depend on interfaces they do not use.
   - **Explanation:** Create specific interfaces for each type of client instead of one general-purpose interface. This keeps the clients isolated from changes in the interfaces they don't use.

```java
// Bad Example
interface Worker {
    void work();
    void eat();
}

class Robot implements Worker {
    @Override
    void work() { /* code to work */ }

    @Override
    void eat() {
        throw new UnsupportedOperationException("Robots don't eat");
    }
}

// Good Example
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}
```

```java
class Robot implements Workable {
    @Override
    void work() { /* code to work */ }
}

class Human implements Workable, Eatable {
    @Override
    void work() { /* code to work */ }

    @Override
    void eat() { /* code to eat */ }
}
```

we may have many light bulbs

5. D - Dependency Inversion Principle (DIP)
   - **Definition:** High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.
   - **Explanation:** To achieve this, depend on interfaces or abstract classes instead of concrete classes and instantiate dependencies via inversion of control (IoC) like dependency injection.

```java
class LightBulb {
    void turnOn() { /* code to turn on */ }
    void turnOff() { /* code to turn off */ }
}

class Switch {
    private LightBulb bulb;

    Switch(LightBulb bulb) {
        this.bulb = bulb;
    }

    void operate() {
        bulb.turnOn();
    }
}

// Good Example
interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable {
    @Override
    void turnOn() { /* code to turn on */ }
    @Override
    void turnOff() { /* code to turn off */ }
}
```

]→ abs depending on detail

depends on LB - for body... → detail

* Array used to create custom arraylist is of Object type ∴ Cast it before fetching or adding...

* Generics: allows you to write classes interface and methods with a Placeholder for the type of data to operate...
  - ↳ <Integer> → wont allow other

* Wildcard: used in generics to represent an unknown type.
  - ↳ Unbounded wildcard (T)
    - * when you dont care abt type
  - ↳ Upper Bounded wildcard (T extends Type)
    - * Type must subtype of specific type eg. Number
  - ↳ Lower Bound wildcard (T extends Type)
    - * Super type → Integer etc...

* Constructor → Default / Parametrized

* Virtual → virtual void display() { — — }
  - ↳ can be instantiated
  - ↳ derived class can override it

* Pure virtual functions
  - ↳ virtual void display() = 0;
  - ↳ makes f'n abstract
  - ↳ must be overriden by derived class