OOPs.

1) Encapsulation.

a: Immutable class (Encapsulation + final + No Setters)

Thread safe.
Cannot be modified.

private final int id.

b) Serializable, transient. (Saving objects)

convert into byte. not include serialization.

Eg:- class user implements Serializable.
{
private String username;
private String password;
}
→ private transient String password.

c) Reflection breaks the OOPs protection. (frameworks)

2) INHERITANCE.

a) Single inheritance
class Z extends X, Y { } //X class C extends B //✓

b) Constructor chaining.

c) Method overriding vs hiding (static methods are hidden, not overridden)

d) Final method cannot be overriden

e) Private Member are not inherited

f) Object class is superclass of All classes.

g) Shadow variables (super, this)
parent to child.

h) IS A Relationship kulayega Inheritance.

i) Upcasting & downcasting.

```
Dog d = new Dog();
Animal a = d; // upcasting
```

```
Animal a = new Dog();
Dog d = (Dog) a; // downcasting
```

```
Animal a = new Animal();
Dog d = (Dog) a; // Error Runtime
```

Generics :-

* Java removes all generic type information at runtime is called Type Erasure.

Meaning :- at compile time, generic help enforces type safety. at runtime, the type becomes raw (list) - so you can't overload based on generic types

Eg :- public void show (List<String> list) { }
public void show (List<String> list) { } X Compile ki

why? public void show (List list) // after type erasure.

Wildcard :-

? extends T

Eg :- List<? extends Number> list = new ArrayList<Integer>();
list.add(10); X can't add anything
list.add(null); // only null is allowed

Number n = list.get(0); // ✓
Integer i = list.get(0); // ✓ (casted)

? super ~~Int~~ T

Eg :- List<? super Integer> list = new ArrayList<Number>();
list.add(10); // ✓
list.add(new Integer(5)); // ✓

Object obj = list.get(0); // allowed ✓
Integer val = list.get(0); // Compiler doesn't allow.

⇒ Because the compiler doesn't know if it's a Number, Object etc. - it's too generic to trust!

Note.

? extends T ⇒ I have a box of some subtype of T : I can look, but I can't add anything to it.

? super T ⇒ I have a box that accepts T or anything above it, I can safely put T into it, but not read specific types from it.

HashCode & Equals.

1) Eg:- String s1 = "FB"; // s1.hashCode() = 2236
String s2 = "Ea"; // s2.hashCode() = 2236.
sop(s1.hashCode() == s2.hashCode()); // true.

```
HashMap<String, Integer> map = new HashMap<>();
map.put(s1, 100);
map.put(s2, 200);
sop(map); // Both key exist
```

hashCode() helps find the buckets.
equals() check inside the bucket for actual key identity.

2) How HashMap resolves Collisions.

```
int hash = key.hashCode();
int index = hash % capacity;
```

Then, in that index (bucket):
if empty – add entry.
if not empty – check existing key via equals.
if key already exists — ~~same~~ overwrite value.
else — chain the new node (linked list / tree).

Eg:- HashMap<key, value>

[Bucket[5] → [key1 = val1] → [key2 = val2]

if key1.hashCode() == key2.hashCode() but !key1.equals(key2), both are stored.

3) TreeMap uses compareTo() for Sorting, Not hashCode().

4) Key Overwriting in HashMap: equals() + hashCode()

Same HashCode, different equals ⇒ Stored separately.
Same hashCode, same equals ⇒ Value is replaced

// If you override equals(), you must override hashCode() -
else unexpected behaviour in HashMap.
// Equals() behaviour.

Collection

→ ArrayList.remove(int index) vs ArrayList.remove(Object o)

Ex: ArrayList<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
list.remove(2); → it takes as the ele at index 2. So, it remove 3.

∴ list.remove(Integer.valueOf(2)); // it removes the ele 2

⇒ TreeMap & TreeSet ⇒ add null throw NPE

HashSet & HashMap ⇒ use this if nulls are needed.

⇒ If two objects have same hashcode() → checkEquals()
If equals() returns true, it's a duplicate & won't add
Set Duplicates not allowed.

⇒ Natural Ordering (Comparable) → Implement Comparable<T> → Default order in TreeSet, TreeMap → ele1.compareTo(ele2)
↳ compareTo

Custom Ordering (Comparator<T>) → You define when creating the collection → compare(ele1, ele2).

TreeMap & TreeSet require sorting.

compareTo()/compare() = 0 means duplicate, Even if 2 obj are not equals()

* foreach loop → use Iterator. → Direct list.remove() → ConcurrentModificationException.

Eg:-
```
for(String s : list){
    if(s.equals("n")){
        list.remove(s);
}}
```

* throw & throwes mismatch give compile time error for checked exception.

try → catch → finally → return / throw

* Only checked exception needs throws like IOException, SQLException

Rule :- If finally contains a return, it wins - earlier return or throw gets ignored.

(catch A | B e) // Compile error.

```
try {  step1
       Step 2
       throw X
   }
catch (Y) {
   step 3
   }
finally {
   step 4
   }
```

Throw X → looks for catch block of type X/parent

No match ? → skip to finally, then crash.

Match found ? → catch runs → finally runs → proceed

* If finally has a throw or return, it overrides whatever was in try.

* throws clause must match the actual ~~expression~~ exception or its parent.

legal.

Eg :-

```
void m3() throws Parent
{ throw new child()
}
```

```
void m2() throws superclass
{ throw new subclass()
}
```

```
void m() throw sibling (IOException)
{ throw new Exception();
}
```

Compile time Error

Polymorphism.

It allows one reference to point to objects of diff types and invoke the right behaviour at runtime.

a) Method Overloading ≠ True Polymorphism.
   - Static
   - resolve at ~~runtime~~ compile time.

b) Method Overriding = True Polymorphism.
   - Overriding via upcasting
   - runtime

c) Method Overriding rules.
   i) Access Modifier → Child class ka modifier bada hona chaiye. parent se

   ii) Return type → can be covariant.

   iii) Exception Rule → Subclass method can throw narrower or same checked exceptions.

d) Polymorphism with Interfaces.
   Interference reference → powerful abstraction

e. Polymorphism & Arrays.
```
Animal[] animals = new Dog[3]; // covariant arrays allow.
animals[0] = new Dog(); // ✓
animals[1] = new Cat(); // ✗ Runtime ArrayStoreException
```

f. Private / Static / Final methods are not overridden

g) instanceof :- used to check whether an object is an instance of specific class, subclass or interface
   return ~~Err~~ boolean type

   Null safety :- null instanceof ~~subtype~~ → ✗ false (won't throw exception)

ABSTRACTION.

It can be achieved by:-
- Abstract classes.
- Interfaces

1) Abstract:-

a) You can define constructors inside abstract classes. constructors runs when subclass is instantiated.

b) Abstract class can have both abstract & non abstract methods

c) Abstract class can have instance variables & static variables.

d) You can declare static & final methods (BUT not abstract static)

//abstract static void print(){} //X not allowed.

e) Can implement interfaces.

f) Can extend concrete class.

g) Can be partially implemented.

h) Can't be final or private.

// final abstract class X{} //X

// private abstract class X{} //X

INTERFACE

a) All variables are public static final — constants.
   ↳ can't make them private, protected / non final.

b) All methods are public abstract (by default)

c) You cannot create object of an interface.

d) From Java 8. interface can have default methods, static methods.

```
default void greet(){
    System.out.println("Hello");
}
```

Calling - Java.super.greet();

e) Methods cannot be private or protected.

f) Supports multiple Inheritance.

g) An interfaces can extend multiple interfaces.

```
interface A {}
interface B {}
interface C extends A,B { } // valid.
```

* class extend only 1 class but interface can extends many interfaces.

h) Interfaces can't have constructors.

i) It is used to represent capability, not structure

MARKERS INTERFACES
   ⇒ Empty interfaces, no methods.
   ⇒ SERIALIZABLE, CLONEABLE, REMOTE

Encapsulation.

Hides data (variables)

Security concern

Achieved using private, public, getter/setters.

focus: Who can access my data?

Abstraction.

Hides implementation (logic)

design concern.

Achieved using abstract class, interface

focus: What should be exposed?

Final Classes & Final Methods.

Final class — cannot be inherited.
Final method — cannot be overriden.
Final variable — constant

Where to use HAS-A (Association, Aggregation, Composition)

Association.

Use:- Two classes are related but can work independently.

Eg:- Doctor ↔ Patient, Student ↔ College.

```
class Doctor{
    Patient assignedPatient;
}
```

// used in loose relationships where no ownership is required

Aggregation.

One object "has-a" another, but they can live separately.

Eg:- Department has Teachers, but Teacher can change dept.

```
class Teacher { }

class Department {
    List<Teacher> teachers;
}
```

// used when you want reusability of the child object.

Composition.

One object completely owns another.

Eg:- Library has Books, Car has Engine, Human has Heart

```
class Engine{}
class Car{
    private Engine engine = new Engine();
}
```

Used when the parent class is responsible for child's lifecycle.

SOLID Principles -

1. Single Responsibility Principle.
   Each class should do one thing & do it well.
   otherwise, it becomes harder to maintain & test.

2. Open/Closed Principle (OCP)
   S/w entities should be open for extension, but closed for modification.

3. You should be able to add new features without changing existing code.

3. Liskov Substitution Principle (LSP)
   Objects of a superclass should be replaceable with objects of its subclass without breaking the app.

idea :- A child class must behave in a way that does not surprise / break the expectations from the parent class

4. Interface Segregation Principle (ISP)
   Clients should not be forced to depend on interfaces they do not use.

idea - Big interfaces into smaller ones.

5. Dependency Inversion Principle (DIP)
   Depend on abstraction, not on concrete classes.

idea - High level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).