6) What is the order of invokation of constructors in inheritance? Is it same as order of constructor execution & object creation?

7) How to pass parameters & custom input to parent class refer variables, ie. parameterized super constructor?

8) Can we write super constructor call anywhere in the child class constructor?

```java
class User {
    String name;
    String location;

    public User() {
        this.name = "Anonymous";
        this.location = "India";
        System.out.println(x: "Guest User Created");
    }

    public void viewShow() {
        System.out.println(x: "I can view listing shows on app");
    }
}
```

```java
class RegisteredUser extends User {
    String emailId;
    long phoneNo;

    public RegisteredUser() {
        this.emailId = "registeredUser@gmail.com";
        this.phoneNo = 9319117888l;
        System.out.println(x: "Registered User Created");
    }

    public void bookShow() {
        System.out.println(x: "I can book the shows on app");
    }
}
```

super();

RegisteredUser user = new RegisteredUser();

Constructor Invocation / Calling

1. RegisteredUser (Child)
2. User (Parent)

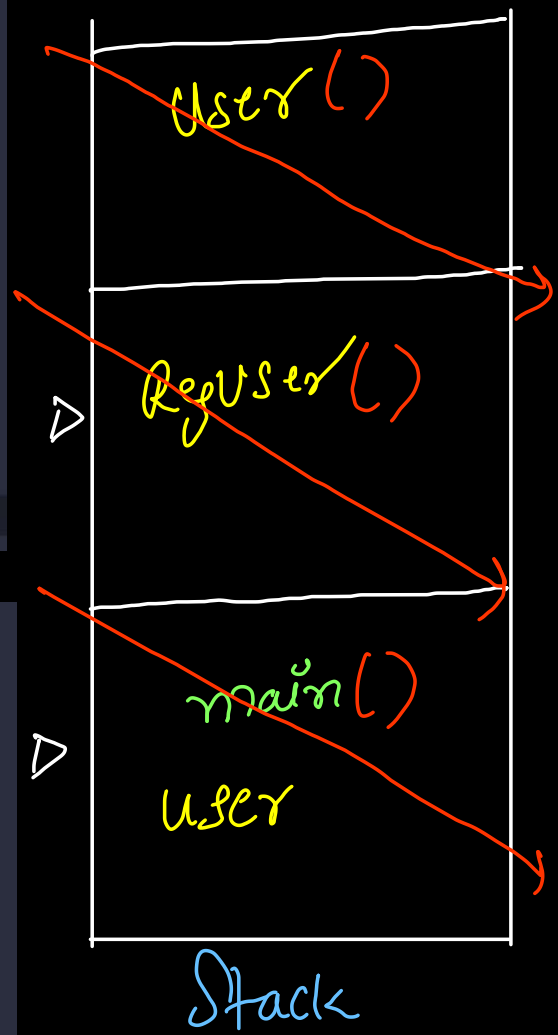
Constructor Execution

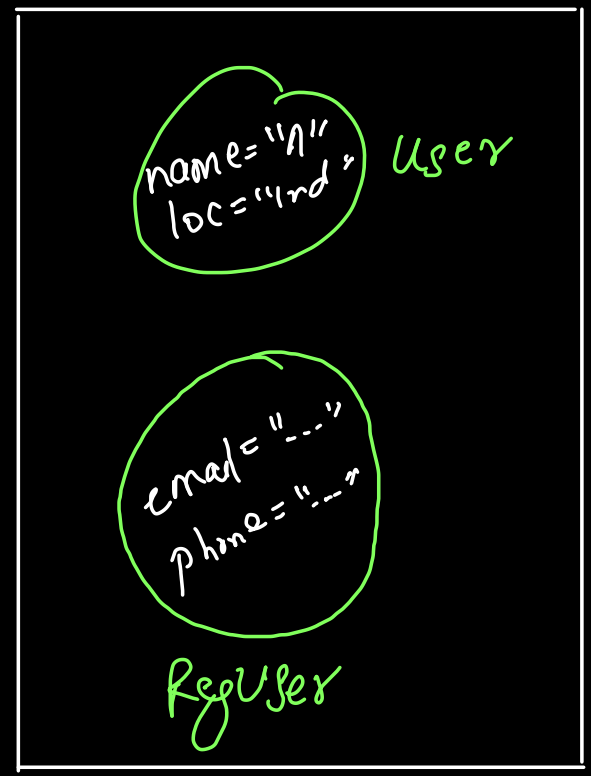
1. User (Parent)
2. RegisteredUser (Child)

```java
class User {
    String name;
    String location;

    public User() {
        this.name = "Anonymous";
        this.location = "India";
        System.out.println(x: "Guest User Created");
    }

    public void viewShow() {
        System.out.println(x: "I can view listing shows on app");
    }
}
```

```java
class RegisteredUser extends User {
    String emailId;
    long phoneNo;

    public RegisteredUser() {
        this.emailId = "registeredUser@gmail.com";
        this.phoneNo = 9319117888l;
        System.out.println(x: "Registered User Created");
    }

    public void bookShow() {
        System.out.println(x: "I can book the shows on app");
    }
}
```

super(); ✓





① Guest User Created (Parent)

② RegUser Created (Child)

```java
class User {
    String name;
    String location;

    public User() {
        this.name = "Anonymous";
        this.location = "India";
        System.out.println(x: "Guest User Created");
    }

    public User(String name, String location) {
        this.name = name;
        this.location = location;
    }
```

```java
RegisteredUser user3 = new RegisteredUser(emailId: "archit.aggarwal@gmail.com",
phoneNo: 9319117889l);
System.out.println(user3.name);
System.out.println(user3.location);
System.out.println(user3.emailId);
System.out.println(user3.phoneNo);

RegisteredUser user4 = new RegisteredUser(name: "archit", location: "Delhi",
emailId: "archit@gmail.com", phoneNo: 9319117889l);
System.out.println(user4.name);
System.out.println(user4.location);
System.out.println(user4.emailId);
System.out.println(user4.phoneNo);
```

```java
class RegisteredUser extends User {
    String emailId;
    long phoneNo;

    public RegisteredUser() {
        // super();
        this.emailId = "registeredUser@gmail.com";
        this.phoneNo = 9319117888l;
        System.out.println(x: "Registered User Created");
    }

    public RegisteredUser(String emailId, long phoneNo) {
        // super();
        this.emailId = emailId;
        this.phoneNo = phoneNo;
    }

    public RegisteredUser(String name, String location, String emailId, long phoneNo) {
        super(name, location);
        this.emailId = emailId;
        this.phoneNo = phoneNo;
    }
```

```
Anonymous
India
archit.aggarwal@gmail.com
9319117889
archit
Delhi
archit@gmail.com
9319117889
```

Q) What is the OBJECT CLASS in Java? Give the methods provided by it.

→ Root of the class heirarchy in Java, ie. topmost class.

→ If not specified, every class (both already defined or user defined) will have parent class as Object class.

→ Object class is parent class for every class directly or indirectly, ie every class will inherit Object class methods by default.

→ It can be used to refer to any class' object (whose type we don't know) (Object class reference variable, child class object)

↳ This is old fashioned way to implement "Generic Programming"

Important Object class methods

(1) protected Object clone() throws CloneNotSupportedException

→ Creates & return copy of this object.

→ By default : shallow copy

→ class overiding clone() method must implement Cloneable interface to tell JVM.

→ Object class does not implement cloneable, calling clone() method Object class will lead to Exception.

(2) protected void finalize() throws Throwable

→ called by Garbage Collector for objects that are not referenced by any one

(3) public boolean equals (Object obj)

→ Indicates whether other object is equal to this one.

→ By default (object class) : compares on addresses!

→ Overriding can be done on some/all properties equality.

Properties :→

(1) Reflexive : a equals a

(2) Symmetric : a equals b (⇐) b equals a

(3) Transitive : a equals b & b equals c ⇒ a equals c

(4) Consistent : a equals b remains same if a & b remain same

(5) a.equals(null) = false

```java
class Movie {
    String name;
    int duration;
    double rating;

    Movie(String name, int duration, double rating) {
        this.name = name;
        this.duration = duration;
        this.rating = rating;
    }

    @Override
    public boolean equals(Object other) {
        if (this == other)
            return true;
        return this.name.equals(((Movie) other).name);
    }
}
```

without overriding

true
false
false

default equals logic

with overriding

true
false
true

custom equals logic

```java
Movie m1 = new Movie(name: "Endgame", duration: 180, rating: 4.9);
Movie m2 = m1;
System.out.println(m1.equals(m2)); // Object's are same: true

Movie m3 = new Movie(name: "Infinity War", duration: 150, rating: 4.7);
System.out.println(m1.equals(m3));

Movie m4 = new Movie(name: "Endgame", duration: 200, rating: 5.0);
System.out.println(m1.equals(m4));
```

```java
System.out.println(m1.equals(m1)); // Reflexive
System.out.println(m2.equals(m1)); // Symmetric
System.out.println(m2.equals(m4)); // Transitive
System.out.println(m1.equals(other: null)); // False
```

④ public final Class getClass()

→ Can't be overriden

→ Returns runtime class of an object. "Class" object of this object can be used to get metadata of this class

Eg Code

```
Object obj1 = new Object();
Class cobj1 = obj1.getClass();
System.out.println("Object 1 : " + obj1 + " ClassName: " + cobj1.getName());

Object obj2 = new String(original: "Hello World");
Class cobj2 = obj2.getClass();
System.out.println("Object 2 : " + obj2 + " ClassName: " + cobj2.getName());

Object obj3 = new Movie();
Class cobj3 = obj3.getClass();
System.out.println("Object 2 : " + obj3 + " ClassName: " + cobj3.getName());
```

Output

```
Finished in 98 ms
Object 1 : java.lang.Object@511d50c0 ClassName: java.lang.Object
Object 2 : Hello World ClassName: java.lang.String
Object 2 : Movie@1d44bcfa ClassName: Movie
```

(5) public int hashcode()

→ Returns a hash code value for the object.
- hash code → unique integer value which is used to uniquely identify the object.
- it does not store/return the address of object

It is used to search objects in collections such as HashSet, HashMap, etc.

Default Behavior :- Every different Object (different memory address) will get unique hashcode.

→ Hashcode for a particular object will remain same forever, ie there cannot be two hashcodes for same object during a single run of application.

Note :- i) If two objects are equal (acc to equals()) method, then they must give same hashcode values.

ii) If two objects are unequal, good hash function should give different hash codes for those objects.

⑥ public String toString()

{ return getClass.getName() + "@" + Integer.toHexString(hashCode()); }

→ Returns String Representation of object

Default Behavior: classname @ unsigned hexadecimal hashcode

Other methods related to Synchronization :→

- public final void notify()
- public final void notifyAll()

→ lock released/thread out of critical section

- public final void wait()
- public final void wait(long timeout)
- public final void wait(long timeout, int nanos)

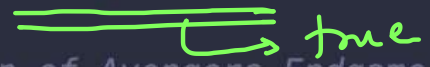
→ lock acquired/thread going in critical section!

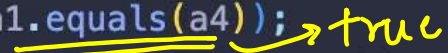
```java
class Movie implements Cloneable {
    String name = "Avengers Endgame";
    int duration = 180;
    double ratings = 4.5;

    public Movie() {
        System.out.println(x: "Object Created - Initialization by Constructor");
    }

    public Movie(String name, int duration, double ratings) {
        this.name = name;
        this.duration = duration;
        this.ratings = ratings;
    }

    @Override
    public boolean equals(Object other) {
        return this.name.equals(((Movie) other).name);
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    @Override
    public String toString() {
        return ("Name : " + this.name + " , Duration : "
                + this.duration + " , Ratings : " + this.ratings);
    }

    @Override
    protected void finalize() throws Throwable {
        // Deallocation of Resources Holded by Object
        // (Database Connection, File Input/Output Streams, H/W Resources like Camera)
        // Similar to Destructors in C++
        // Automatically Called By Garbage Collector Daemon Thread handled by JVM
        System.out.println(x: "Object Destroyed - Resources Clean Up By Finalize");
    }

    @Override
    public int hashCode() {
        // Custom Hashing Logic
        return this.name.hashCode();
    }
}
```

```java
@SuppressWarnings("rawtypes")
public static void getClassDemo() {
    Object obj1 = new Object();
    Class cobj1 = obj1.getClass();
    System.out.println("Object 1 : " + obj1 + " ClassName: " + cobj1.getName());

    Object obj2 = new String(original: "Hello World");
    Class cobj2 = obj2.getClass();
    System.out.println("Object 2 : " + obj2 + " ClassName: " + cobj2.getName());

    Object obj3 = new Solution();
    Class cobj3 = obj3.getClass();
    System.out.println("Object 3 : " + obj3 + " ClassName: " + cobj3.getName());
}
```

```
architaggarwal@Archits-MacBook-Air 01. Core Java Basics % java Solution
Object 1 : java.lang.Object@136432db ClassName: java.lang.Object
Object 2 : Hello World ClassName: java.lang.String
Object 3 : Solution@7382f612 ClassName: Solution
```

```java
public static void equalsDemo() {
    Movie a1 = new Movie(name: "Avengers Endgame", duration: 180, ratings: 4.9);

    // Different Movies
    Movie a2 = new Movie(name: "Avengers InfinityWar", duration: 150, ratings: 4.8);
    System.out.println(a1.equals(a2)); // False

    // Exactly Same Movie (Addresses/References are Same)
    Movie a3 = a1;
    System.out.println(a1.equals(a3)); // True (Address Comparison First)

    // Extended Version of Avengers Endgame
    Movie a4 = new Movie(name: "Avengers Endgame", duration: 200, ratings: 5.0);
    System.out.println(a1.equals(a4));
    // By Default (Without Overriding : Object's equals) : False
    // With Overriding and Name Comparison (Movie's equals) : True
}
```





```java
public static void cloneDemo() throws Exception {
    Movie a1 = new Movie(name: "Avengers Endgame", duration: 180, ratings: 4.9);
    System.out.println(a1);

    Movie a2 = (Movie) (a1.clone());
    System.out.println(a2);
}
```

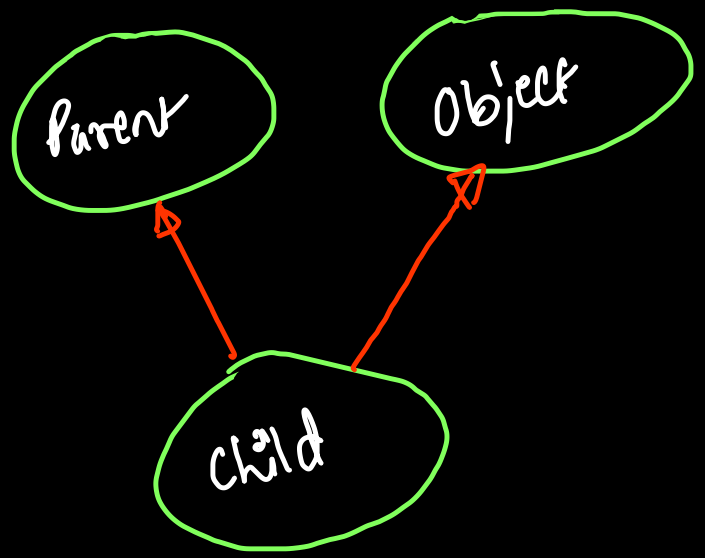
```
architaggarwal@Archits-MacBook-Air 01. Core Java Basics % java Solution
Name : Avengers Endgame , Duration : 180 , Ratings : 4.9
Name : Avengers Endgame , Duration : 180 , Ratings : 4.9
```

To tostring methods

```java
public static void hashCodeDemo() {
    Movie a1 = new Movie(name: "Avengers Endgame", duration: 180, ratings: 4.9);
    System.out.println(a1.hashCode());

    Movie a2 = new Movie(name: "Avengers InfinityWar", duration: 150, ratings: 4.8);
    System.out.println(a2.hashCode());

    Movie a3 = a1;
    System.out.println(a3.hashCode());

    Movie a4 = new Movie(name: "Avengers Endgame", duration: 200, ratings: 5.0);
    System.out.println(a4.hashCode());
}
```

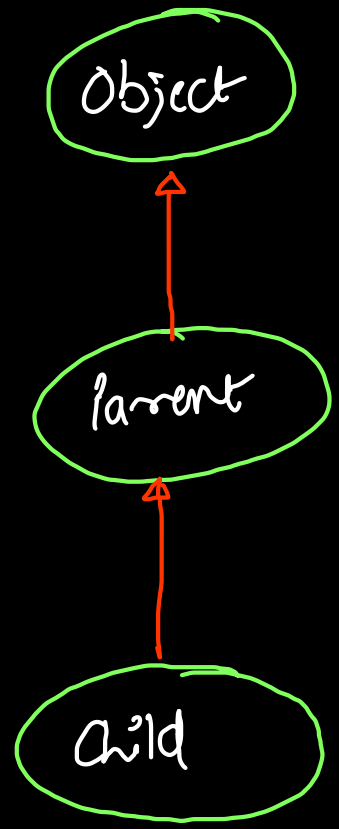
```
architaggarwal@Archits-MacBook-Air 01. Core Java Basics % java Solution
-2084956566
1213999165
-2084956566
-2084956566
```

Q) If Object class is a parent class for Every class, so it means there might be multiple inheritance possible?



Multiple Inheritance(✗)

Hybrid Inheritance(✗)



multilevel inheritance!

Q) What do you mean by POLYMORPHISM in Java? What are the different types of Polymorphism? What are the advantages & disadvantages of Polymorphism?

Polymorphism :- Performing single action (behavior → method) in many different ways.

many form

Two types →
- Compile time or Static Polymorphism : method overloading
- Run time or Dynamic Polymorphism : method over-riding : Dynamic method Dispatch

Advantages

- Code reusability is the main advantage of polymorphism; once a class is defined, it can be used multiple times to create an object.
- In compile-time polymorphism, the readability of code increases, as nearly similar functions can have the same name, so it becomes easy to understand the functions.
- The same method can be created in the child class as in the parent class in runtime polymorphism.
- Easy to debug the code. You might have intermediate results stored in arbitrary memory locations while executing code, which might get misused by other parts of the program. Polymorphism adds necessary structure and regularity to computation, so it is easier to debug.

Disadvantages

- Implementing code is complex because understanding the hierarchy of classes and its overridden method is quite difficult.
- Problems during downcasting because implicitly downcasting is not possible. Casting to a child type or casting a common type to an individual type is known as downcasting.
- Sometimes, when the parent class design is not built correctly, subclasses of a superclass use superclass in unexpected ways. This leads to broken code.
- Runtime polymorphism can lead to the real-time performance issue (during the process), it basically degrades the performances as decisions are taken at run time because, machine needs to decide which method or variable to invoke.

Q) What do you mean by Compile-time / static polymorphism or method overloading in Java. Give some real world examples.

↳ two or more methods in the same class with same function-name, and different argument list

Rules for Overloading

1. Number of arguments should be different
   or
2. Order of arguments should be different
   or
3. Type of argument should be different

Change in return type does not matter!

⇒ determined by compiler: which function call to be binded with which method definition.

```java
class Sum {
    public void sum(int a, int b) {
        System.out.println(a + b);
    }

    // Number of Arguments
    public void sum(int a, int b, int c) {
        System.out.println(a + b + c);
    }

    // Datatypes of Arguments
    public void sum(String a, String b) {
        System.out.println(a + b);
    }

    public void sum(String a, int b) {
        System.out.println(a + b);
    }

    // Order of Arguments
    public void sum(int a, String b) {
        System.out.println(a + b);
    }
}
```

def^n body

```java
class Driver {
    Run | Debug
    public static void main(String[] args) {
        Sum obj = new Sum();

        obj.sum(a: 10, b: 15);
        obj.sum(a: 5, b: 10, c: 15);
        obj.sum(a: "Archit", b: "Aggarwal");
        obj.sum(a: "Archit's score: ", b: 100);
        obj.sum(a: 100, b: " is Archit's score");
    }
}
```

call (invocation)

```
25
30
ArchitAggarwal
Archit's score: 100
100 is Archit's score
```

```java
// Compilation Error: Variable names
// does not matter
// public void sum(int c, int d){
// }

// Compilation Error: Return type does
// not matter
// public int sum(int a, int b){
// return a + b;
// }
```

Re-declaration Error (Compilation)

↓

overloading

```java
class User {
    String name;

    public void setName(String firstName, String
    middleName, String lastName) {
        name = firstName + " " + middleName + " " +
        lastName;
    }

    public void setName(String firstName, String
    lastName) {
        name = firstName + " " + lastName;
    }

    public void setName(String firstName) {
        name = firstName;
    }

    public void setName(char firstLetter, char
    secondLetter, String lastName) {
        name = firstLetter + ". " + secondLetter +
        ". " + lastName;
    }
}
```

```java
User u1 = new User();
u1.setName(firstName: "Sachin",
middleName: "Ramesh", lastName: "Tendulkar");
System.out.println(u1.name);

User u2 = new User();
u2.setName(firstName: "Virat",
lastName: "Kohli");
System.out.println(u2.name);

User u3 = new User();
u3.setName(firstName: "Tejas");
System.out.println(u3.name);

User u4 = new User();
u4.setName(firstLetter: 'K',
secondLetter: 'L', lastName: "Rahul");
System.out.println(u4.name);
```

Real World Example

```
Sachin Ramesh Tendulkar
Virat Kohli
Tejas
K. L. Rahul
```

8) What do you mean by run-time/dynamic polymorphism or method overriding in Java. Give some real world examples.

↳ One method in parent class, another method in child class, with same function prototype

- same function name
- same return type *(TRC conditions)
- Same argument list.
  - same no of arguments
  - same order of arguments
  - datatype of arguments same

In runtime, JVM binds the function call to the corresponding function definition

There must be inheritance for over-riding; ie. two methods in different classes related as parent-child.

```java
class Parent {
    private int a, b;

    public int getA() {
        return a;
    }

    public void setA(int a) {
        this.a = a;
    }

    public int getB() {
        return b;
    }

    public void setB(int b) {
        this.b = b;
    }

    // overriden method
    public void printObject(int a, int b) {
        System.out.println(x: "Parent's Object : ");
        this.setA(a);
        this.setB(b);
        System.out.println(this.getA() + " " + this.getB());
    }
}
```

```java
class Child extends Parent {
    private int p, q;

    public int getP() {
        return p;
    }

    public void setP(int p) {
        this.p = p;
    }

    public int getQ() {
        return q;
    }

    public void setQ(int q) {
        this.q = q;
    }

    // overrided method
    public void printObject(int p, int q) {
        System.out.println(x: "Child Object : ");
        this.setP(p);
        this.setQ(q);
        System.out.println(super.getA() + " " + super.getB());
        System.out.println(this.getP() + " " + this.getQ());
    }
}
```

```java
public static void main(String[] args) {
    Parent obj1 = new Parent();
    obj1.printObject(a: 10, b: 20);
    // overrident method -> Parent

    Child obj2 = new Child();
    obj2.printObject(p: 40, q: 50);
    // overrided method -> Child
}
```

```
Parent's Object :
10 20
Child Object :
0 0
40 50
```

```java
class User {
    String name;
    String location;

    public User(String name, String location) {
        this.name = name;
        this.location = location;
    }

    // overriden method
    public void bookShow() {
        System.out.println(this.name + " " + this.location);
        System.out.println(x: "Error: You cannot book the show");
        System.out.println(x: "Please, first login or sign up");
    }
}
```

```java
class RegisteredUser extends User {
    String emailId;
    long phoneNo;

    public RegisteredUser(String name, String location, String emailId,
    long phoneNo) {
        super(name, location);
        this.emailId = emailId;
        this.phoneNo = phoneNo;
    }

    // overrided method
    public void bookShow() {
        System.out.println(super.name + " " + super.location + " " +
        this.emailId + " " + this.phoneNo);
        System.out.println(x: "Please select the number of seats");
        System.out.println(x: " And proceed to payment gateway");
    }
}
```

```java
User u1 = new User(name: "Archit", location: "Delhi");
u1.bookShow();

RegisteredUser u2 = new RegisteredUser(name: "Archit",
location: "Delhi", emailId: "archit@gmail.com",
phoneNo: 9319117889l);
u2.bookShow();
```

```
Archit Delhi
Error: You cannot book the show
Please, first login or sign up
Archit Delhi archit@gmail.com 9319117889
Please select the number of seats
 And proceed to payment gateway
```

```
class Child extends Parent {
    private int p, q;

    public int getP() {
        return p;
    }

    public void setP(int p) {
        this.p = p;
    }

    public int getQ() {
        return q;
    }

    public void setQ(int q) {
        this.q = q;
    }
```

```
Parent's Object :
10 20
Parent's Object :
40 50
```

} If child does not have overriding method

Q) What are the differences between method overloading and method overriding?

Method Overloading

(by Javac compiler binding)

1. implements compiletime polymorphism
2. methods are statically binded (method call determined at compile time)
3. methods must be in same class
4. methods must have different argument list.
5. Return type may or may not be same

Method Over-riding

(by JVM binding)

1. implements runtime polymorphism
2. methods are dynamically binded (method call determined at run-time)
3. methods must be in super & sub class
4. methods must have same signature (same return type & argument-list)
5. Return type (*) must be same
   → co-variant type

Q) What do you mean by Static binding. What do you understand by dynamic binding. What are the differences? What do you mean by dynamic method dispatch?

```java
class Movie {
    int duration = 180;
    String name = "Avengers Endgame";

    public void display() {
        System.out.println(this.name + " runs for " +
        this.duration);
    }
}

class Driver {
    Run | Debug
    public static void main(String[] args) {
        Movie avengers = new Movie();
        avengers.display();
    }
}
```

Static binding {compiler}

no polymorphism

Static Binding

→ binding of function call to the corresponding function definition at compile-time by compiler is known as static binding.

→ It is done for methods which are not over-ridden and may/may be overloaded.

→ Also known as early binding

→ Static, final, private methods are always statically binded.

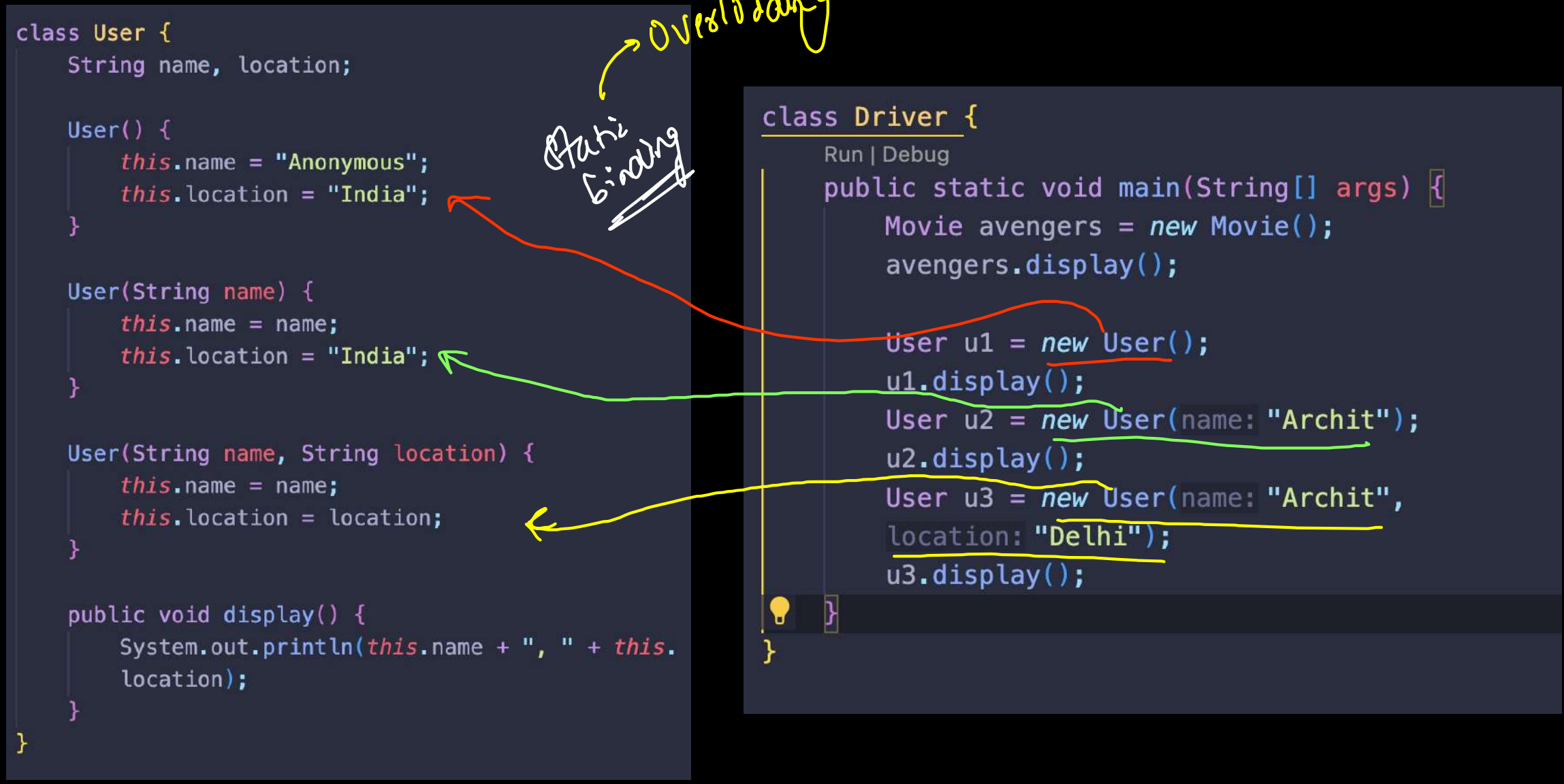
Dynamic Binding

→ binding of function call to the corresponding function definition at runtime by JVM is known as dynamic binding.

→ It is done for methods which are over-ridden in the child class ⇒ dynamic method dispatch!

→ Also known as late binding

→ Abstract methods are always late binded.


Overloading
Static binding
class User {
String name, location;
User() {
this.name = "Anonymous";
this.location = "India";
}
User(String name) {
this.name = name;
this.location = "India";
}
User(String name, String location) {
this.name = name;
this.location = location;
}
public void display() {
System.out.println(this.name + ", " + this.
location);
}
}
class Driver {
Run | Debug
public static void main(String[] args) {
Movie avengers = new Movie();
avengers.display();
User u1 = new User();
u1.display();
User u2 = new User(name: "Archit");
u2.display();
User u3 = new User(name: "Archit",
location: "Delhi");
u3.display();
}
}

```java
class RegisteredUser extends User {
    String phone = "9319117889";

    @Override
    public void display() {
        System.out.println(this.name + ", " + this.
        location + ", " + this.phone);
    }
}
```

dynamic binding of runtime polymorphism & overriding

```java
RegisteredUser u4 = new RegisteredUser();
u4.display();
```

```java
class A{
    public void earlyBind(){
        System.out.println("Early Bind");
    }

    public void lateBind(){
        System.out.println("Late Bind in Parent Class");
    }
}

class B extends A{
    @Override
    public void lateBind(){
        System.out.println("Late Bind in Child Class");
    }
}

public class Main{
    public static void main(String[] args){
        A obj = new B();
        obj.earlyBind();→ Early
        obj.lateBind();→ Child
    }
}
```

Q) What are other forms of polymorphism in Java?

- Implicit operator overloading
- Coersion (implicit or explicit type conversion)

We cannot explicitly achieve operator overloading in Java

- complex number addition

c1 = 5+3i
c2 = 7+9i
} ⊕ ✗
add() ✓

```java
public static void main(String[] args) {
    int var1 = 100;
    int var2 = 200;

    System.out.println(var1 + var2);
    // + => Addition of two integers

    String str1 = "Archit";
    String str2 = "Aggarwal";
    System.out.println(str1 + str2);
    // + => Concatenation of String

    System.out.println(var1 + var2 + str1);
    // Addition then Concatenation

    System.out.println(str1 + var1 + var2);
    // Concatenations only
    System.out.println(var1 + str1 + var2);

    System.out.println(x: "\t Archit \n Aggarwal");
}
```

+ operator overloaded

```
300
ArchitAggarwal
300Archit
Archit100200
100Archit200
        Archit
 Aggarwal
```

\ escape sequences → overloading

Coercion

```java
public static void fun(long var) {
    System.out.println(var);
}
```

```java
fun(var: 999999999991); // long -> long
fun(var: 200); // int -> long
fun(var: 'A'); // char -> long
```

} implicit type conversion
↳ upcasting
smaller data → big data

```java
public static void fun2(char var) {
    System.out.println(var);
}
```

```java
fun2((char) 100l); // long -> char
fun2((char) 35); // int -> char
fun(var: 'A'); // char -> char
```

} explicit type conversion
↳ downcasting
big data → smaller data

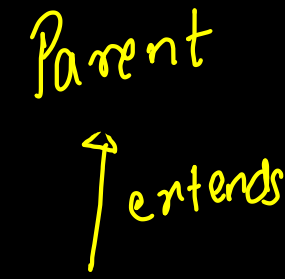
Q) which of the following are correct declarations. Why or why not?

(a) `Parent obj1 = new Parent();` valid

(b) `Child obj2 = new Child();` valid

* (c) `Parent obj3 = new Child();` valid

(d) `Child obj4 = new Parent();` not valid



Q) What are Polymorphic Variables in Java? Give some real-world examples.

```java
class User {
    String name, address;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public User(String name, String address) {
        this.name = name;
        this.name = address;
    }

    public void bookShow() {
        System.out.println(x: "You can't book show.
        Please register first");
    }
}
```

override

```java
class RegisteredUser extends User {
    String phoneNo, emailId, address;

    public String getPhoneNo() {
        return phoneNo;
    }

    public void setPhoneNo(String phoneNo) {
        this.phoneNo = phoneNo;
    }

    public String getEmailId() {
        return emailId;
    }

    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public RegisteredUser(String name, String address, String phoneNo, String
    emailId) {
        super(name, address: "Delhi");
        this.phoneNo = phoneNo;
        this.emailId = emailId;
        this.address = address;
    }

    public void bookShow() {
        System.out.println(x: "You can book the show, please proceed to payments");
    }
}
```

GuestUser (Parent)

name, address (city)

getters & setters

bookshow()

{ syso("can't"); }

extends

RegisteredUser (Child)

phoneNo, emailID, address (complete address)

getters & setters

bookshow()

{ syso("can"); }

GuestUser (Parent)

name, address → city, bookshow()

getters & setters

{ Syso("can't"); }

extends

Registered User (Child)

phoneNo, emailID, address → complete address

bookshow()

getters & setters

{ Syso("can"); }

```
GuestUser User = new GuestUser();
```

GuestUser ↑ reference variable

new GuestUser(); ↑ object/instance

```java
User u1 = new User(name: "Archit",
address: "Delhi");

System.out.println(u1.getName() + " " + u1.
getAddress());

u1.bookShow();

System.out.println(u1.name + " " + u1.
address);
```

```
Archit Delhi
You can't book show. Please register first
Archit Delhi
```

GuestUser (Parent)

name, address — accessible (indirectly) super

getters & setters

bookshow() — accessible (indirectly) super

{ syso("can't"); }

extends

RegisteredUser (Child)

phoneNo, emailID, address

getters & setters

bookshow()

{ syso("can"); }

RegiUser User = new RegiUser();

reference variable

object/instance

user.address

child ✓

parent ✓

super.address ✓

```
RegisteredUser u2 = new RegisteredUser
(name: "Archit", address: "Delhi 110085",
phoneNo: "9319117889", emailId: "archit.
aggarwal023@gmail.com");

System.out.println(u2.name + ", " + u2.
emailId + ", " + u2.phoneNo);

u2.bookShow(); // overriding
System.out.println(u2.address);
System.out.println(u2.getAddress());
```

"can book the show"

this hides parent's address variable

```
Archit, archit.aggarwal023@gmail.com, 9319117889
You can book the show, please proceed to payments
Delhi 110085
Delhi
Delhi 110085
```

super.address (indirect)

this.address (direct access)

GuestUser (Parent)

name, address

getters & setters

overriden

bookshow()

{ syso("can't"); }

@override

extends

RegisteredUser (Child)

{created but not accessible}

phoneNo, emailID, address

getters & setters

overrided

bookshow()

{ syso("can"); }

GuestUser User = new RegisteredUser();

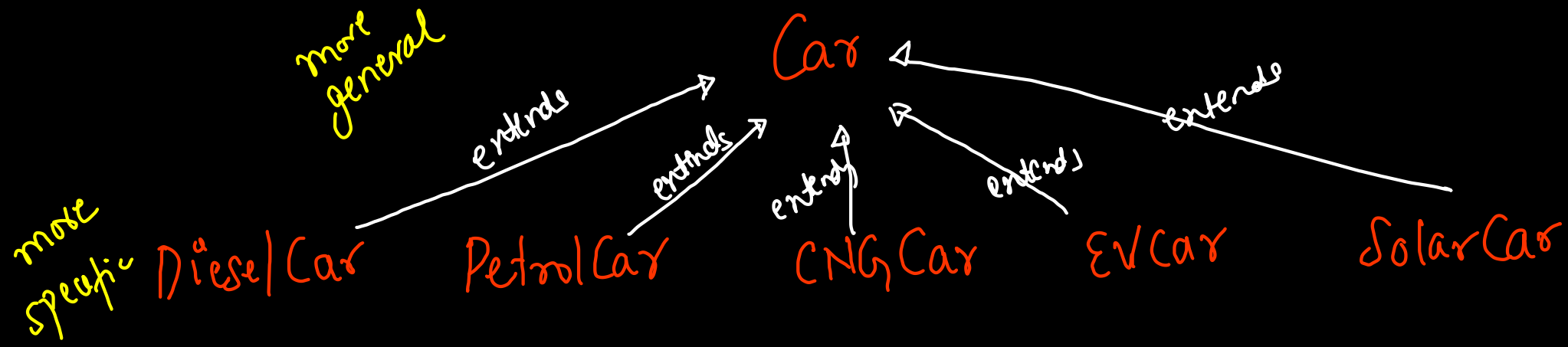
↑ reference variable → variables & func^n prototype

↑ object/instance → function body

Polymorphic Variables

⟹ Reference variables pointing to different objects are known as polymorphic variables.

↑ parents variable



```
DieselCar c1 = new DieselCar();
PetrolCar c2 = new PetrolCar();
CNGCar c3 = new CNGCar();
EVCar c4 = new EVCar();
SolarCar c5 = new SolarCar();
```

This is not preferred

```
Car c1 = new DieselCar();
Car c2 = new PetrolCar();
Car c3 = new CNGCar();
Car c4 = new EVCar();
Car c5 = new SolarCar();
```

Car is a polymorphic variable

RegisteredUser u4 = new GuestUser( );

emailId & phoneNo

name, address

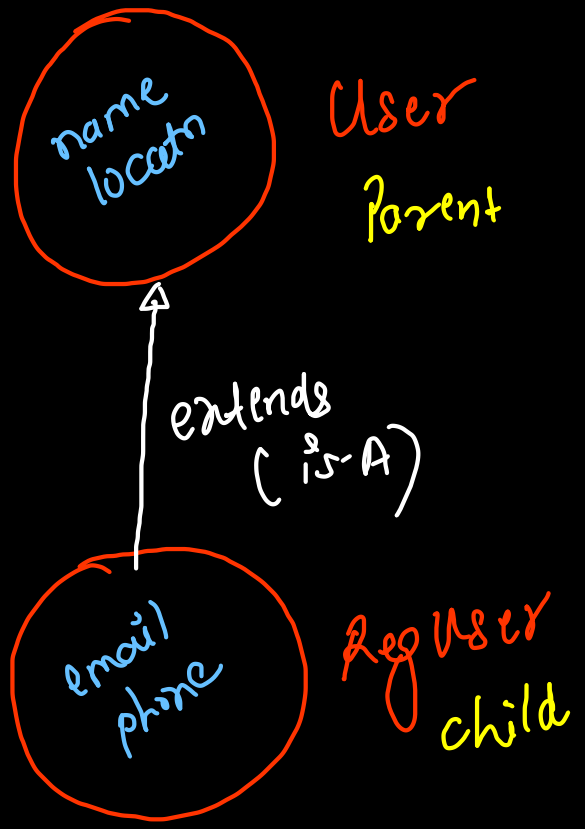
GuestUser( ) —child→ subclass

RegUser()

constructor

Q) What do you mean by upcasting and downcasting? Out of both, which is done implicitly and which needs to be done explicitly. Give some coding examples.

Q) What is instanceof operator and when to use it? What are the advantages of using it?




```
User u1 = new User();                    ✓

RegUser u2 = new RegUser();              ✓ allowed

User u3 = new RegUser();                 ✓ typecasting
Parent      child
upcasting implicit

wrong { RegUser u4 = new User(); }
        child        Parent

RegUser u5 = u2; // shallow copy allowed

RegUser u6 = u3; // compilation error downcasting

RegUser u6 = (RegUser)u3; // allowed

RegUser u7 = u1; // compilation error

u7 = (RegisteredUser)u1; // runtime error
```

```java
public static void main(String[] args) {
    RegisteredUser u1 = new RegisteredUser(name: "archit", location: "delhi",
    emailId: "archit@gmail.com", phoneNo: 9319117888l);
    System.out.println(u1.name + " " + u1.emailId);

    // Child: Ref & Object

    User u2 = new User(); // Parent : Ref & Object
    System.out.println(u2.name);

    User u3 = new RegisteredUser(); // Parent Ref, Child Object
    System.out.println(u3.name + ((RegisteredUser) u3).emailId);

    // RegisteredUser u4 = new User(); child ref parent object not allowed

    RegisteredUser u5 = u1; // shallow copy
    System.out.println(u5.name + " " + u5.emailId);

    // RegisteredUser u6 = u3; // Compilation Error: No Typecasting
    if (u3 instanceof RegisteredUser) {
        RegisteredUser u6 = (RegisteredUser) u3;
        System.out.println(u6.name + u6.emailId);
    } else {
        System.out.println(x: "This user is not registered");
    }

    // RegisteredUser u7 = u2; // Compilation Error
    // RegisteredUser u7 = (RegisteredUser) u2;
    // Runtime Error: Classcast Exception

    if (u2 instanceof RegisteredUser) {
        RegisteredUser u7 = (RegisteredUser) u2;
        System.out.println(u7.emailId);
    } else {
        System.out.println(x: "This user is not registered");
    }
}
```

→ explicit down casting

```
archit archit@gmail.com
Guest User Created
Anonymous
Guest User Created
Registered User Created
AnonymousregisteredUser@gmail.com
archit archit@gmail.com
AnonymousregisteredUser@gmail.com
This user is not registered
```

Q) What do you mean by method hiding? Give some examples.

- overriding static methods
- overriding private methods
- change in parameters in overided methods

How is it different from method over-riding?

```
Parent{
    method()
}
```

```
Child extends Parent{
    method()
}
```

```java
class Parent {
    int parentData;

    public static void staticFun() {
        System.out.println(x: "This is parent's static function");
    }
}

class Child extends Parent {
    int childData;

    public static void staticFun() {
        System.out.println(x: "This is child's static function");
    }
}
```

```java
class Driver {
    @SuppressWarnings("all")
    Run | Debug
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        obj1.staticFun();

        Child obj2 = new Child();
        obj2.staticFun();

        Parent obj3 = new Child();
        obj3.staticFun(); // Parent's Fun

        // No Dynamic Method Dispatch: No overriding
    }
}
```

Static Functions

Overriding ✗

Hiding ✓

dynamic binding ✗

static binding ✓

→ Parent

→ Child

→ Parent

```java
class Parent {
    int parentData;

    public static void staticFun() {
        System.out.println(x: "This is parent's static function");
    }

    public void privateFun() {
        System.out.println(x: "This is parent's private function
        but it is public");
    }

    public void publicFun() {
        System.out.println(x: "This is parents's fun with 0
        parameter");
    }
}
```

```java
class Child extends Parent {
    int childData;

    public static void staticFun() {
        System.out.println(x: "This is child's static function");
    }

    // private void privateFun() {}
    // Parent Public -> Child Private

    public void publicFun(int data) {
        System.out.println(x: "This is child's fun with 1
        parameter");
    }
}
```

```java
class Driver {
    @SuppressWarnings("all")
    Run | Debug
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        obj1.staticFun();
        obj1.publicFun();
        // obj1.publicFun(10); // THis is not allowed

        Child obj2 = new Child();
        obj2.staticFun();

        obj2.publicFun();
        obj2.publicFun(data: 10);
        // Overloading with functions in different classes

        Parent obj3 = new Child();
        obj3.staticFun(); // Parent's Fun
        obj3.publicFun(); // Parent's Fun

        // No Dynamic Method Dispatch: No overriding
    }
}
```

Q) Why data members cannot be over-rided? What do you understand by variable hiding in Java?

Instance variable hiding refers to a state when instance variables of the same name are present in superclass and subclass.Now if we try to access using subclass object then instance variable of subclass hides instance variable of superclass irrespective of its return types.

Q) Can constructors be overrided? Give reasons.

No

accessible ↑ super()

We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).

Q) Can overrided methods have different return types? What do you mean by covariant types?

The covariant return type specifies that the return type may vary in the same direction as the subclass. Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type

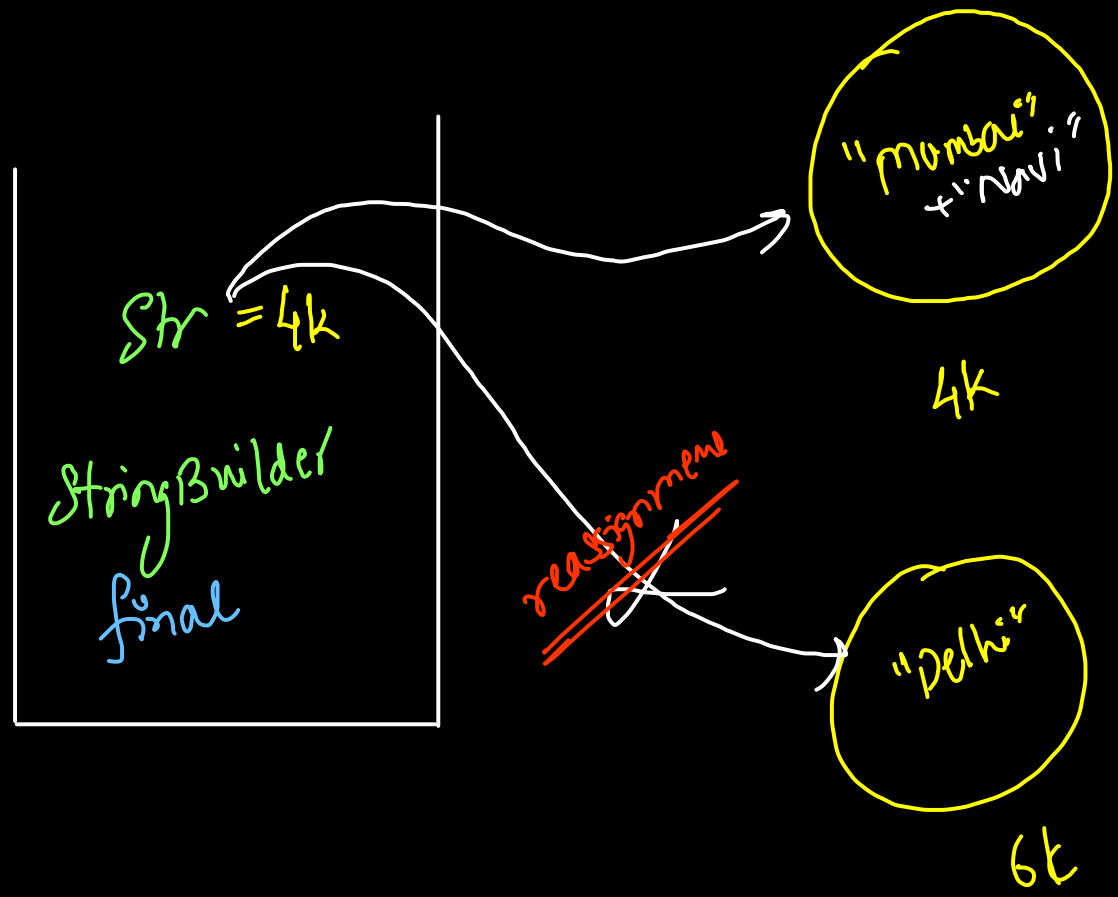
```java
class Parent {
    public void fun() {
        System.out.println(x: "This is parent's void fun");
    }

    public Parent getObject() {
        System.out.println(x: "This is parent's GetObject");
        return new Parent();
    }
}

class Child extends Parent {
    // public int fun(){
    // System.out.println("This is parent's int fun");
    // }
    // Invalid: Child should have void returntype

    @Override
    public Child getObject() {
        System.out.println(x: "This is child's GetObject");
        return new Child();
    }
    // Non-Primitive Covariant Type
}
```

```java
class Driver {
    Run | Debug
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        Child obj2 = new Child();

        Parent obj3 = obj1.getObject();
        obj3.fun();
        // Parent Ref: Parent Object

        // Child obj4 = obj1.getObject(); // Not Valid

        Parent obj5 = obj2.getObject(); // Overriding
        obj5.fun();
        // Parent Ref: Child Object

        Child obj6 = obj2.getObject();
        obj6.fun();
        // Child Ref: Child Object
    }
}
```

```
This is parent's GetObject
This is parent's void fun
This is child's GetObject
This is parent's void fun
This is child's GetObject
This is parent's void fun
```

Q) What are the applications of final keyword?

- final primitive variables : constants → CAPITAL_LETTERS, Snake {underscore}
- final reference variables : Reference can't be changed but data (instance) can be.
- final methods : can't be over-riden (to stop runtime polymorphism)
  - ↳ Static Binding
- final class : can't be extended (to stop inheritance)
  - // // inherited
  - ↳ A final class will automatically have all its functions final

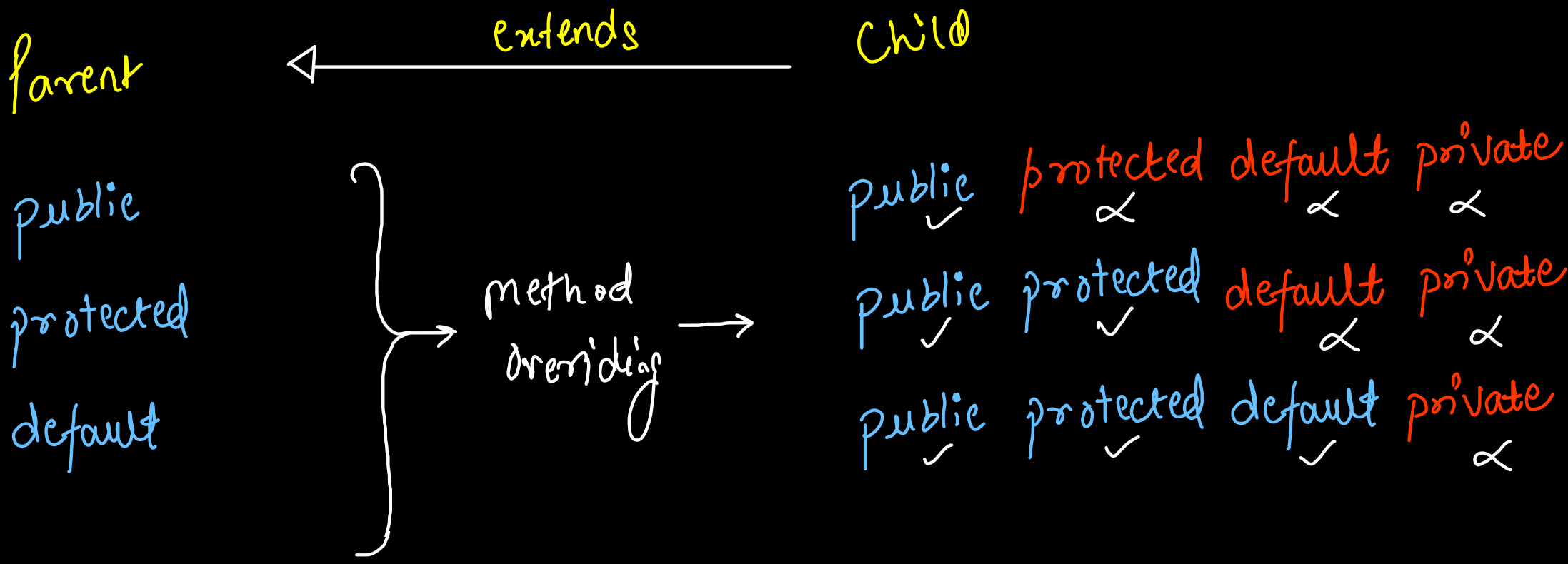



```java
class Driver {
    private static final double PI = 3.14;
    private static final StringBuilder str = new
    StringBuilder(str: "Mumbai");

    Run | Debug
    public static void main(String[] args) {
        // Get: Constant Variable: Allowed
        System.out.println(Driver.PI);

        // Set: Constant Variabel: Not Allowed
        // Driver.PI = 22/7.0;
        // Reassignment not possible

        System.out.println(Integer.MIN_VALUE);
        System.out.println(Integer.MAX_VALUE);

        // Get And Data Modification: Final Reference
        Variable
        System.out.println(str);
        str.append(str: " Navi");
        System.out.println(str);

        // Reassignment Not Allowed
        // str = new StringBuilder("Delhi");

    }
}
```

```java
class Parent {
    public final void finalFun() {
        System.out.println(x: "This is parent's fun");
    }
}

class Child extends Parent {
    // Final Method can't be overrided
    // public void finalFun(){}
}

final class Parent2 {
    public void finalFun() {
        System.out.println(x: "This is parent2's fun");
    }
}

// Final Class cannot be extended
// class Child2 extends Parent2{}
```

Access Modifiers & Method Overriding

Q) Explain the statement "Overrided method (child class) must be less restrictive than overriden method (parent class)"



Parent {

public void fun() {

}

}

extends

child {

private void fun() {

}

}

→ should be accessible within child class

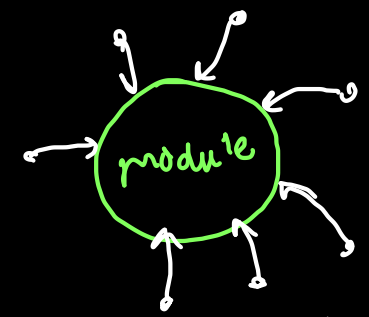
Parent obj = new Child();

obj.fun(); //over-riding → Dynamic method Dispatch

allowed to access private method outside the class

Q) What do you mean by ABSTRACTION in Java? What are the advantages? How to achieve/implement abstraction?

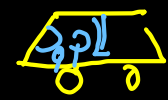
→ Complexity ↓, redundancy ↓, consistency ↑



complexity ↓
essential info ↑↑

Data abstraction is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either abstract classes or interfaces.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.



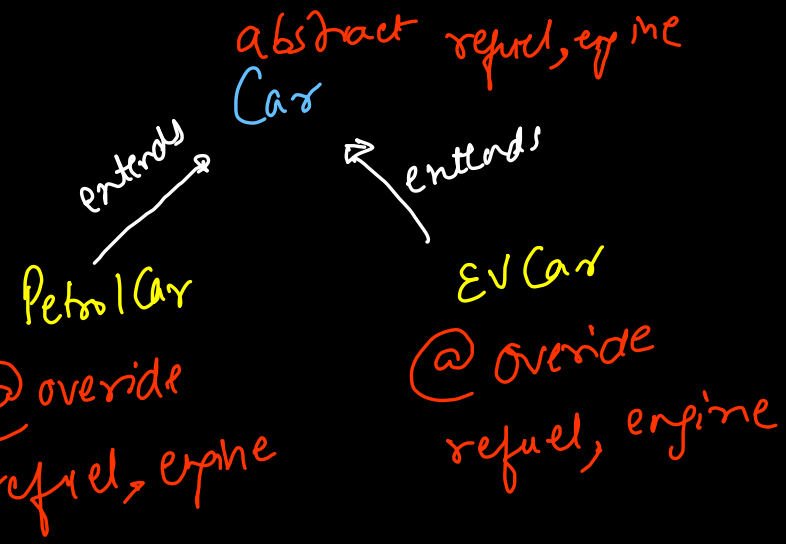
Q) What is an abstract class? What are abstract methods?
How is abstract class different from concrete class?

- abstract class cannot be instantiated
- abstract class may have some abstract methods.
  - method with only function prototype (access modifier, return type, function name, argument list) but no body
- abstract class is a more generic class whereas concrete class is a more specific class.
- Concrete class (implementation) depends upon abstract class (abstraction)

```java
abstract class Car {
    abstract void refuel();

    abstract void engine();
}

class PetrolCar extends Car {
    @Override
    void refuel() {
        System.out.println(x: "Petrol Refill");
    }

    @Override
    void engine() {
        System.out.println(x: "It has a Petrol
        Engine");
    }
}

class EVCar extends Car {
    @Override
    void refuel() {
        System.out.println(x: "Battery recharge");
    }

    @Override
    void engine() {
        System.out.println(x: "Spark/Electricity
        based engine");
    }
}
```

generic class { doesn't exist in real life }

Specialized class




```java
class Driver {
    Run | Debug
    public static void main(String[] args) {
        // Car obj = new Car();
        // We cannot create objects of Car (abstrac
        class)

        PetrolCar obj = new PetrolCar();
        obj.refuel();
        obj.engine();

        EVCar obj2 = new EVCar();
        obj2.refuel();
        obj2.engine();
    }
}
```

```java
class Driver {
    Run | Debug
    public static void main(String[] args) {
        // Car obj = new Car();
        // We cannot create objects of Car (abstract
        class)

        PetrolCar obj = new PetrolCar();
        obj.refuel();
        obj.engine();

        EVCar obj2 = new EVCar();
        obj2.refuel();
        obj2.engine();

        // Polymorphism
        Car c1 = new PetrolCar();
        c1.refuel();
        System.out.println(c1.color);

        Car c2 = new EVCar();
        c2.refuel();
        System.out.println(c2.color);
    }
}
```

```
Petrol Refill
It has a Petrol Engine
Battery recharge
Spark/Electricity based engine
Petrol Refill
Red
Battery recharge
Red
```

Q) Can abstract class have (a) zero (b) some (c) all methods as abstract?

(a) zero ⇓ yes ↙ all concrete methods

(b) some ⇓ yes (0% – 100%)

(c) all ⇓ yes ↓ all abstract methods

Q) Can there be an abstract method inside a concrete (non-abstract) class?

NO, concrete class will throw compilation error
abstract method can be defined in abstract class only.

Q) Can abstract methods be (a) final (b) static (c) private?

abstract ↓ overriding ✓

(a) final ↓ Overriding ✗ ⇒ no

(b) static ↓ method hiding ✓ overriding ✗ early binded ⇒ no

(c) private ↓ early binded ✓ overriding ✗ ⇒ no

Q) Constructors & abstract classes :-

(a) can there be constructors inside a abstract class? Yes
↳ to instantiate child class objects & constructor chaining super();

(b) can constructor itself be abstract? (overriding ✓) (overriding ✗) abstract constructors (no)

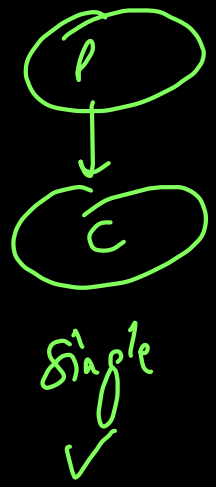
(c) Does abstract class have "this" keyword?
↳ yes → due to object creation of child

```java
abstract class Car {
    String color;

    public Car() {
        color = "White";
    }

    public Car(String color) {
        this.color = color;
    }

    abstract void refuel();

    // static Abstract,
    // private abstract,
    // final abstract
    // These are invalid combinations

    abstract void engine();

    void drive() {
        System.out.println(x: "Drive Car");
    }
}
```

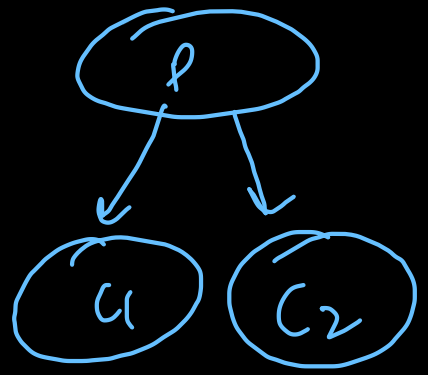
```java
You, 31 seconds ago | 1 author (You)
class PetrolCar extends Car {
    String fuel;

    PetrolCar() {
        super();
        this.fuel = "Petrol";
    }

    PetrolCar(String fuel, String color) {
        super(color); // constructor: Abstract Class
        this.fuel = fuel;
    }

    @Override
    void refuel() {
        System.out.println(x: "Petrol Refill");
    }

    @Override
    void engine() {
        System.out.println(x: "It has a Petrol
        Engine");
    }
}
```

```java
PetrolCar obj3 = new PetrolCar(fuel: "petrol -
Xtra", color: "Black");
System.out.println(obj3.color);→Black
System.out.println(obj3.fuel); → petrol-Xtra
```

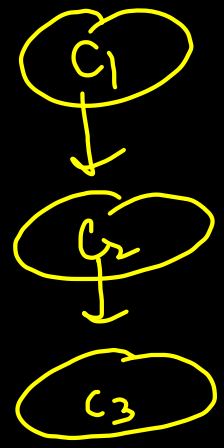
Q) Can we implement multiple inheritance using abstract classes?



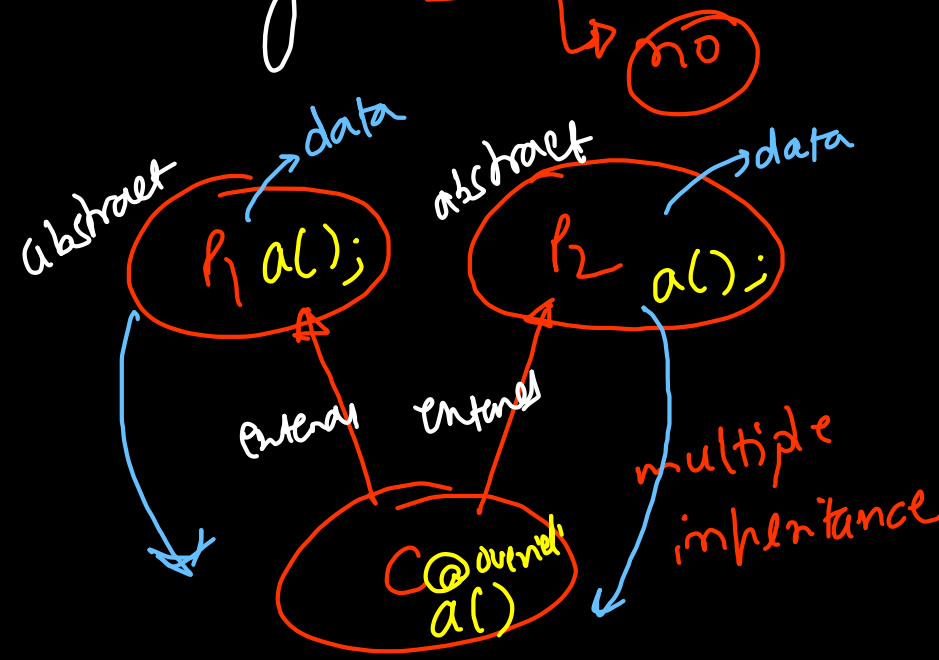
Single ✓



Heirarchial ✓



multilevel ✓



data X 2

↳ ambiguity

multiple inheritance not allowed

Q) What is the difference between encapsulation, data hiding and data abstraction?

Encapsulation :→ Wrapping together properties & behavior in a single entity known as class.

Data Hiding :→ Properties / Behaviors → access modifiers
- public
- default / protected
- private

Abstraction :→ Hiding the Unneccessary details from the client
- abstract class
- interface

Abstraction	Data Hiding
It is the process of hiding the internal implementation and keeping the complicated procedures hidden from the user. Only the required services or parts are displayed.	It is the process that ensures exclusive data access to class members and hiding the internal data from outsiders.
Focuses on hiding the complexity of the system.	Focuses on protecting the data by achieving encapsulation (a mechanism to wrap up variables and methods together as a single unit).
This is usually achieved using abstract class concept, or by implementing interfaces.	This can be achieved using access specifiers, such as private, and protected.
It helps to secure the software as it hides the internal implementation and exposes only required functions.	This acts as a security layer. It keeps the data and code safe from external inheritance as we use setter and getter methods to set and get the data in it.
It doesn't affect end users, since the developers can perform changes internally in implementation classes without changing the abstract method in the interfaces.	This ensures that users can't access internal data without authentication.
It can be implemented by creating a class that only represents important attributes without including background detail.	Getters and setters can be used to access the data or to modify it.

Q) What is an interface in Java? Give some real world coding example.

- way to achieve abstraction in Java.
- blueprint of a class
- interface cannot be instantiated

ITheatre → interface
↓ blueprint of a class
Theater → class
name, seats, pvr cinepolis
↓ blueprint of object
Object

Q) What is the default state of variables in an interface?
↳ instance variables ✗
↳ public, static, final
* * ↳ class property
* ↳ constants

Q) What is the default state of methods in an interface?
↳ public, abstract (100% by default)
* abstraction

```java
interface ITheater {
    String industry = "Bollywood";
    // public, static, final

    // 100% abstraction

    void viewShow(); // public, abstract

    String bookShow();
}

class Theater implements ITheater {
    String name;

    public void viewShow() {
        System.out.println(x: "Only View Access");
    }

    public String bookShow() {
        System.out.println(x: "Book Access");
        return "ticket";
    }
}
```

```java
class BookMyShow {
    @SuppressWarnings("all")
    Run | Debug
    public static void main(String[] args) {
        Theater pvr = new Theater();
        pvr.name = "PVR Cinemas";

        pvr.viewShow();

        System.out.println(ITheater.industry);
        System.out.println(Theater.industry);

        Theater cinepolis = new Theater();
        cinepolis.name = "Cinepolis Experience";

        cinepolis.bookShow();

        System.out.println(pvr.industry);
        System.out.println(cinepolis.industry);

    }
}
```

```
Only View Access
Bollywood
Bollywood
Book Access
Bollywood
Bollywood
```

Q) Can there be concrete methods in interface?

↳ by default, you cannot have concrete methods in Java (version < 8)

exception ↳ private method, static method, default method

↳ private method ↳ concrete

↳ static method ↳ concrete

↳ default method ↳ default implementation

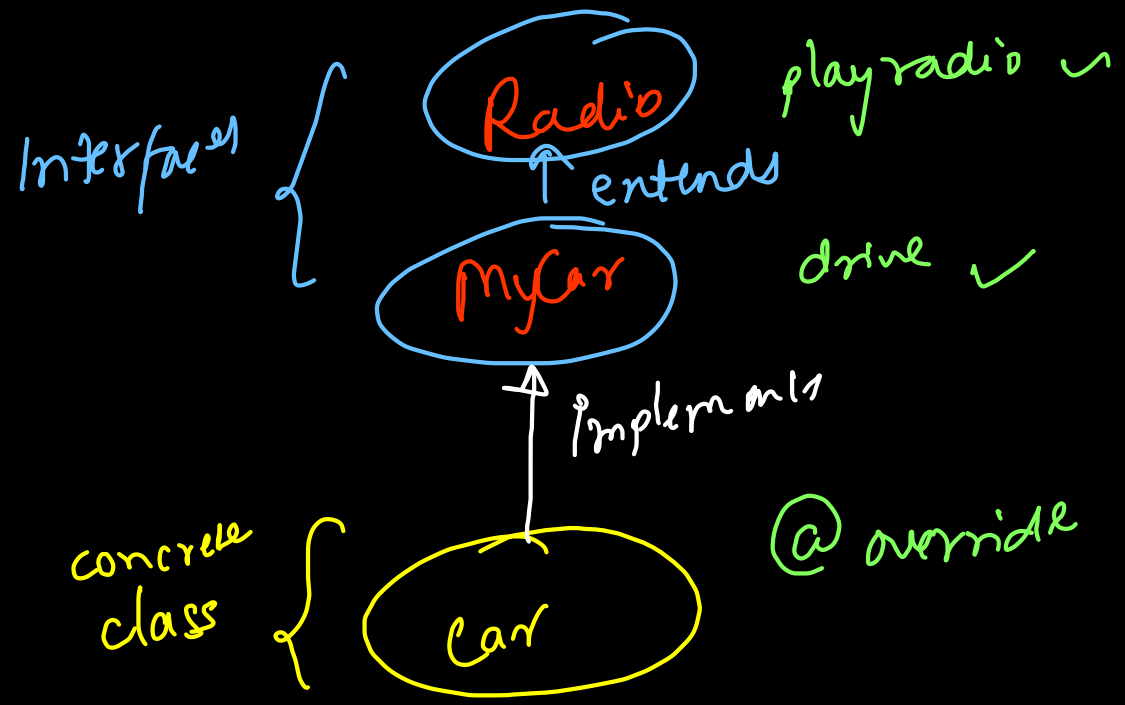
Q) Can an interface be related to another interface. If yes, how?

↳ yes : one interface extends another interface!

```java
interface MyInterface {
    // Default Method: Object's Method
    default void defaultFun() {
        System.out.println(x: "Default fun: have a body");
        privateFun();
    }

    // private method: Within Interface
    private void privateFun() {
        System.out.println(x: "Private fun: have a body");
    }

    // Interface's Method
    public static void staticFun() {
        System.out.println(x: "Static fun: have a body");
    }
}

class MyClass implements MyInterface {
}

class Driver {
    Run | Debug
    public static void main(String[] args) {
        MyInterface.staticFun();

        MyClass obj = new MyClass();
        obj.defaultFun();
    }
}
```

```
Static fun: have a body
Default fun: have a body
Private fun: have a body
```

```java
interface Radio {
    void playRadio();
}

interface MyCar extends Radio {
    void drive();
}

class Car implements MyCar {
    public void playRadio() {
        System.out.println(x: "Radio Starts");
    }

    public void drive() {
        System.out.println(x: "Car Starts");
    }
}

class Driver {
    Run | Debug
    public static void main(String[] args) {
        Car i10 = new Car();
        i10.drive();
        i10.playRadio();
    }
}
```



There is no object of interface ⇒ there is no this keyword

Q) Is there (a) this (b) constructor defined in an interface?

(a) this → no

(b) constructor → no

interface → 100% abstraction

variables → static & final

instance variables ∝ constructor (initializer)

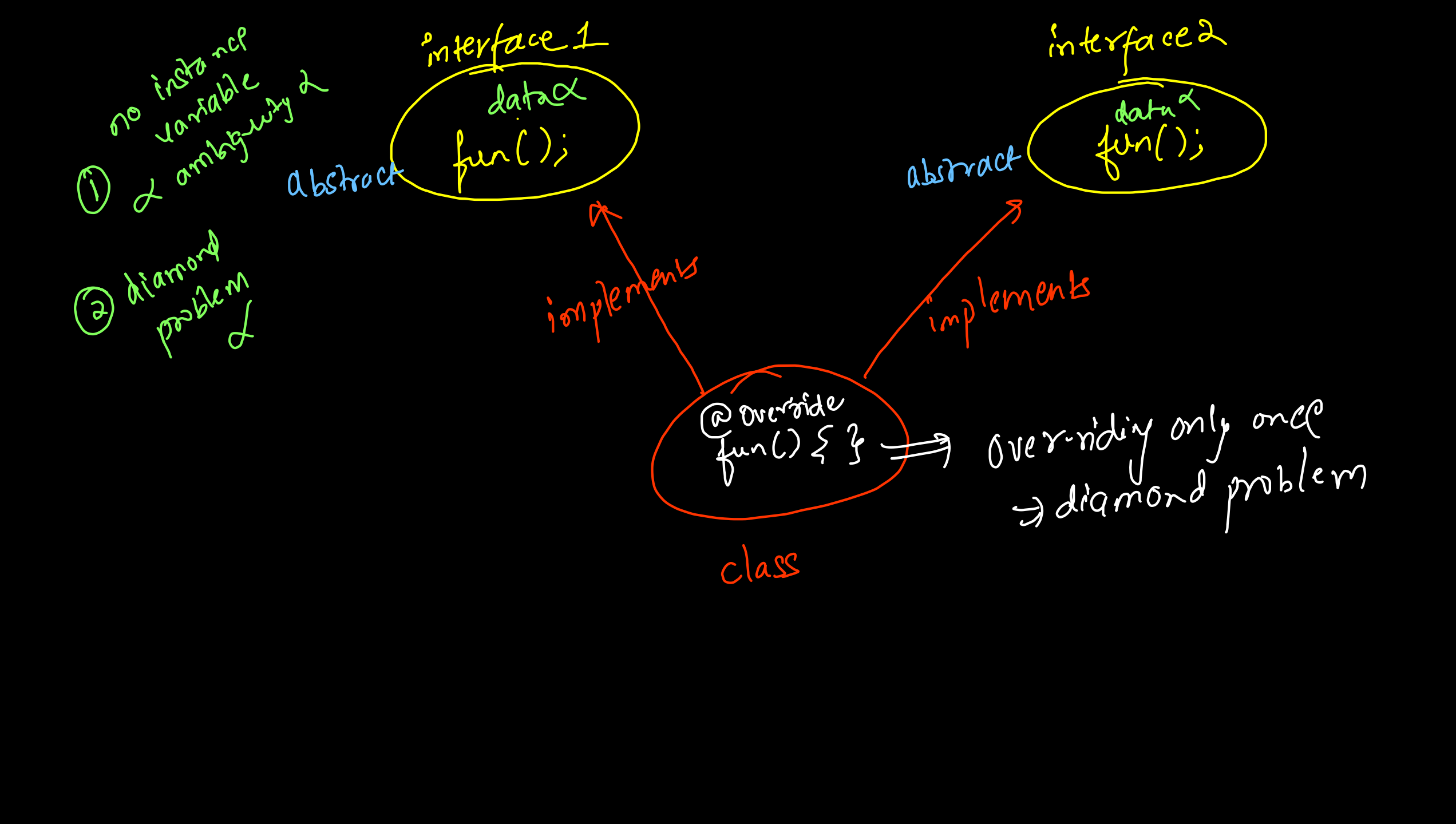
→ interfaces do not take part in class heirarchy

super() → child ↔ Object ✓ Interface ✗

Q) Can class implement more than 1 interfaces? will it not cause diamond problem?

↳ Class can implement 1 or more than 1 interfaces!

{ multiples interfaces can be implemented by a single class }


① no instance variable ∝ ambiguity ∝
② diamond problem ∝
interface 1
data∝
fun();
abstract
interface 2
data∝
fun();
abstract
implements
implements
@Override
fun() { }
class
over-riding only once
⇒ diamond problem

```java
You, 1 second ago | 1 author (You)
interface Radio2 {
    void start();
}

You, 1 second ago | 1 author (You)
interface MyCar2 {
    void start();
}

You, 1 second ago | 1 author (You)
class Car2 implements Radio2, MyCar2 {
    @Override
    public void start() {
        System.out.println(x: "Radio & Car
        Starts Together");
    }
}
```

Abstract

Ambiguity ?

→ No diamond problem

→ only once

```java
Car2 i20 = new Car2();
i20.start();
```

Output

```
You, 1 second ago | 1 author (You)
interface Radio2 {
    String data = "Radio"; // static

    void start();
}

You, 1 second ago | 1 author (You)
interface MyCar2 {
    String data = "Car"; // static

    void start();
}

You, 1 second ago | 1 author (You)
class Car2 implements Radio2, MyCar2 {
    @Override
    public void start() {
        System.out.println(x: "Radio & Car
        Starts Together");
    }

    public void fun() {
        System.out.println(Radio2.data);
        System.out.println(MyCar2.data);
    }
}
```

Static property
of Interface name
↓
Ambiguity

Q) What is the difference between class & interface?

Class

1. blueprint of an object
2. achieve encapsulation
3. it can be instantiated
4. constructors, this, super ✓
5. properties → static/nonstatic nonfinal/final
6. methods → abstract/concrete

Interface

1. blueprint of a class
2. achieve 100% abstraction
3. it cannot be instantiated
4. constructor, this, super ✗
5. properties → public static final
6. all methods are abstract & public

Class	Interface
The keyword used to create a class is "class"	The keyword used to create an interface is "interface"
A class can be instantiated i.e, objects of a class can be created.	An Interface cannot be instantiated i.e, objects cannot be created.
Classes does not support multiple inheritance.	Interface supports multiple inheritance.
It can be inherit another class.	It cannot inherit a class.
It can be inherited by another class using the keyword 'extends'.	It can be inherited by a class by using the keyword 'implements' and it can be inherited by an interface using the keyword 'extends'.
It can contain constructors.	It cannot contain constructors.
It cannot contain abstract methods.	It contains abstract methods only.
Variables and methods in a class can be declared using any access specifier(public, private, default, protected)	All variables and methods in a interface are declared as public.
Variables in a class can be static, final or neither.	All variables are static and final.

Q) What are the differences between abstract classes & interfaces?

Similarities → to achieve abstraction, cannot instantiate

Abstract class	Interface
① 0-100% abstraction	① 100% abstraction
② by default :→ concrete method	② by default :→ abstract method
③ variables → instance variables, final/static variable	③ variables → final/static
④ multiple inheritance (✗)	④ class can implement multiple interfaces
⑤ access modifier → public/default/protected/private	⑤ access modifier → public

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.