Page No.
Date

Class —

- It is a blueprint for creating objects.
- It defines attributes (data members) and behaviors (methods).
- But it does not occupy memory until an object is created.
- Ex: A car class with attributes like color, model, and methods like drive() or brake()

⇒
```
class Car {
    String brand;
    int speed;
    void drive() {
        System.out.println(brand + " is driving at " + speed);
    }
}
```

Object —

- It is an instance of a class.
- It represents a real world entity with state and behavior.
- Each object gets its own copy of instance variables.
- Space is allocated on the heap, and a reference is stored in the stack when an object is created.

⇒
```
public class ObjectDemo {
    public static void main(String[] args) {
        Car c1 = new Car();          //object created
        c1.brand = "Tesla";
        c1.speed = 100;
        c1.drive();
    }
}
```

Page No.
Date

Encapsulation —

- It is the practice of wrapping data (variables) and methods (functions) together in a single unit and restricting direct access to data.
- This is achieved using access modifiers (private, public, protected).
- It ensures data security, prevents accidental modifications and provides controlled access via getters / setters.

⇒

```
class Student {
    private String name;                    // data hidden
    public String getName(){                // getter
        return name;
    }
    public void setName(String name){       // setter
        this.name = name;
    }
}
public class Demo{
    public static void main(String[] args){
        Student s = new Student();
        s.setName("Riya");
        System.out.println(s.getName());
    }
}
```

* Remember Demo class can't access private of Student class ~~therefore, needs getter~~ to access it (setter is also ~~therefore~~ needed to set value).

Page No.
Date

Polymorphism —

- It means "many forms". It allows the same method or operator to behave differently based on context.
- Compile time (overloading) → same method name, different parameters
- Run time (overriding) → subclass provides its own implementation of a method
- It improves flexibility and scalability, letting the same interface be used for different underlying forms.

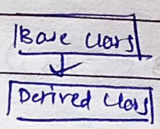
⇒

```
class Calculator {
    int add (int a, int b) {
        return a+b;
    }
    double add ( double a, double b) {
        return a+b;
    }
}
public class OverloadingDemo {
    public static void main ( String[] args) {
        Calculator c = new Calculator();
        System.out.println ( c.add ( 5, 10));
            // int version
        System.out.println ( c.add ( 3.5, 2.5));
            // double version
    }
}
```

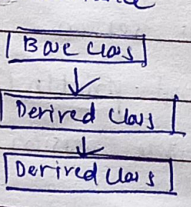
⇒

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
public class OverridingDemo {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound();                    // calls Dog's version
    }
}
```

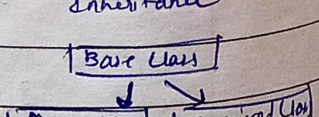
Single Inheritance



Multilevel Inheritance



Hierarchical Inheritance



Page No.
Date

Inheritance —

- It allows one class to acquire the properties and methods of another class.
- It promotes code reusability and creates a "is-a" or parent-child relationship.
- Java support multiple inheritance → Not with classes, But with interfaces
- Ex: A Car class can inherit from a Vehicle class

⇒

```
class Vehicle {
    void start() {
        System.out.println("Vehicle starts");
    }
}
class Car extends Vehicle {
    void honk() {
        System.out.println("Car honks");
    }
}
public class InheritanceDemo {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();          // inherited method
        c.honk();           // own's method
    }
}
```

- Types : Single, Multilevel, Hierarchical
- It does not support multiple inheritance in Java to avoid ambiguity ("diamond problem").

Page No.
Date

Abstraction –

- It hides implementation details and shows only essential features to the user.
- It is achieved using abstract classes or interfaces in Java/Python.
- Encapsulation → how we hide data
  Abstraction → what functionality is exposed.

⇒

```
abstract class Shape {
    abstract void draw();                    // abstract method
    void info() {                            // concrete method
        System.out.println("this is a shape");
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}

public class AbstractionDemo {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.draw();
        s.info();
    }
}
```

```
abstract class Shape {
    Shape() {                                // constructor
        S.O.P("creating new shape");
    }
}
class Circle extends Shape {
    Circle() {
        S.O.P("created a circle");
    }
}
public class oops {
    public static void main(String args[]) {
        Circle c = new Circle();
    }
}
// output :
creating new shape
created a circle
// known as Constructor Chaining
```

Page No.
Date

Abstract class —

- It is a class that cannot be instantiated directly.
- It may have abstract methods (without implementation) and concrete methods (with implementation).
- Used when classes share common behavior but also need their own specific implementation.
- It is prefered over an interface when you want to share common code / fields along with forcing subclasses to implement specific methods.

Interfaces —

- It defines a contract of methods that a class must implement.
- It only has method signatures (in Java <8), but from Java 8 onward it can also have default and static methods.
- Abstract class → can have state (fields) + partial implementation

  Interface → pure contract, multiple inheritance possible

⇒

```
interface vehicle {
    void start();
}
class Bike implements vehicle {
    public void start() {
        System.out.println("Bike starts with a kick");
```

imp to write

```
int headlight = 2;
for example!
// all fields are public, static & final by default
// all methods are public & abstract by default
```

```
    }
}
public class InterfaceDemo {
    public static void main (String[] args) {
        Vehicle v = new Bike();
        v.start();
    }
}
```

Singleton class –

- It ensures that only one instance of a class is created in the entire application, and it provides a global access point to that instance.
- To implement a singleton in Java → make constructor private, create static instance, provide getInstance() method.
- Example : Logger, Database Connection

⇒

```
class Singleton {
    private static Singleton instance;          //single object
    private Singleton() { }                     // private constructor
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    void show() {
```

// For multiple inheritance

```
interface Animal {
    int eyes = 2;
    void walk();
}
interface Herbivore {
    void eat();
}
class Horse implements Animal, Herbivore {
    public void walk(){
        S.O.P ("walks on 4 legs");
    }
    public void eat(){
        S.O.P ("eat veges");
    }
}
public class oops {
    public static void main (String args[]){
        Horse h = new Horse();
        h.walk();
        h.eat();
    }
}
```

```
        System.out.println("Singleton instance working!");
    }
}
public class SingletonDemo {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();
        obj1.show();
        System.out.println(obj1 == obj2);
    }
}
```

Constructor —

- It is a special method used to initialize an object when it's created.
- It has same name as the class and no return type.
- Constructor can be overloaded by having different parameter lists.

⇒

```
class Student {
    String name;
    int age;
    Student(String n, int a) {          // constructor
        name = n;
        age = a;
    }
    void display() {
        System.out.println(name + " is " + age);
```

```
    }
}
public class ConstructorDemo {
    public static void main (String[] args){
        Student s = new Student ("Riya", 20);
            // constructor called
        s.display();
    }
}
```

- 1> copy constructor
  - It creates a new object as a copy of an existing object.
  - Common in C++ (className (const className &obj)), while in Java you typically achieve this by writing a custom constructor or using clone().
  - We use it when we want a deep copy of an object.

⇒

```
# include <iostream>
using namespace std;
class Student {
    string name;
    int age;
public:
    Student (string n, int a) {
        name = n;
        age = a;
```

```
class Student {  int age ; String name;
  public void printInfo() {
      S.O.PL this.name);
      S.O.PL this.age);
  }
  Student (Student s2) {        // copy constructor
      this.name = s2.name
      this.age = s2.age;
  }
  Student () {        //default constructor
  }
}
public class oops {
  public static void main (String args[]) {
      Student s1 = new Student();
      s1.name = "aman";
      s1.age = 24;
      Student s2 = new Student
                   (s1);
      s2.printInfo();
  }
}
```

```
Student ( const Student &s) {
    // copy constructor
    name = s.name ;
    age = s.age ;
}
void display () {
    cout << name << "is" << age <<
    " years old " << endl;
}
};
int main () {
    Student s1 ("Riya", 20);
    Student s2 = s1 ;
    s2.display ();
}
```

- 2) Default Constructor
  - It is a constructor with no parameters, automatically provided by the compiler if none is defined.
  - If you define another constructor, the compiler no longer provides a default one unless you explicitly write it.

⇒

```
class Student {
    String name ;
    int age ;
    Student () {        //default constructor
        name = "Unknown" ;
```

```
        age = 0;
    }
    void display() {
        System.out.println(name + "is" +
                           age);
    }
}
public class DefaultDemo {
    public static void main(String[] args) {
        Student s = new Student();
        // calls default constructor
        s.display();
    }
}
```

- 3) Parameterized Constructor
  - It takes arguments to initialize an object with specific values.
  - We can have multiple constructors (default + parameterized) with different signatures.

⇒

```
class Student {
    String name;
    int age;
    Student(String n, int a) {
        // Parameterized constructor
        name = n;
        age = a;
    }
    void display() {
```

```
        System.out.println(name + "is" +
                                age);
    }
}
public class ParameterizedDemo{
    public static void main(String[] args){
        Student s = new Student("Arjun",
                                21);
        s.display();
    }
}
```

Destructor —

- It is a method that cleans up resources when an object is destroyed.
- Java doesn't have destructor like C++ because java uses automatic garbage collection to manage memory.

⇒

```
#include <iostream>
using namespace std;
class Student {
public:
    Student() {
        cout << "Constructor called" << endl;
    }
    ~Student() {
        cout << "Destructor called" << endl;
```

```
    }
};
int main() {
    Student s1;          // constructor runs
}  // destructor runs here automatically
```

Association –

It is a relationship where two classes are connected but independent.

```
class Teacher {
    string name;
    Teacher(string name) {
        this.name = name;
    }
};
class Student {
    string name;
    Student(string name) {
        this.name = name;
    }
    void learnFrom(Teacher t) {
        System.out.println(name + "learns from" +
                           t.name);
    }
};
public class AssociationDemo {
    public static void main(String[] args) {
```

```
        Teacher t = new Teacher("Mr. Sharma");
        Student s = new Student("Riya");
        s.learnFrom(t);                    // association
    }
}
```

Aggregation –

It is a weaker "has-a" relationship where one class contains another, but the contained object's lifecycle is independent.

⇒

```
class Department {
    String name;
    Department(String name) {
        this.name = name;
    }
}
class University {
    private List<Department> departments;
    University(List<Department> departments) {
        this.department = departments;
    }
    void showDepartments() {
        for(Department d : departments) {
            System.out.println(d.name);
        }
    }
}
```

Page No.
Date

```
public class AggregationDemo {
    public static void main (String[] args){
        Department d1 = new Department ("CS");
        Department d2 = new Department ("Maths");
        List<Department> deptList =
                new ArrayList<>();
        deptList.add (d1);
        deptList.add (d2);
        University u = new University (deptList);
        u.showDepartments ();
    }
}
```

Composition –

It is a "has-a" relationship where one class contains another class as a part, and the contained object's lifecycle depends on the container.

⇒

```
class Engine {
    void start () {
        System.out.println ("Engine starts");
    }
}
class Car {
    private Engine engine;
    Car () {
        Engine = new Engine();
```

```
    }
    void drive() {
        engine.start();
        System.out.println("Car is driving");
    }
}
public class CompositionDemo {
    public static void main(String[] args) {
        Car car = new Car();
        car.drive();
    }
}
```

Hashing —

- It is a technique of mapping data (keys) to a fixed size table (hash table) using a hash function.
- It provides fast access, insertion, and deletion on average O(1)
- Good hash function → uniform distribution, low collisions, deterministic.
- Collision Handling
  - A collision occurs when two different keys produce the same hash value.
  - Collision handling strategies ensure data can still be stored and retrieved correctly.
  - Techniques : Chaining and Open Addressing

Page No.
Date

- 1) Chaining
  - It handles collisions by storing multiple elements at the same index in a linked list (or dynamic array).
  - If multiple keys map to the same hash, they're added to the chain at that bucket.
  - Pros → simple, dynamic size
    Cons → extra memory overhead, degraded performance if chains grow long
- 2) Open Addressing
  - All elements are stored in the hash table itself.
  - On collision, we probe the next empty slot.
  - Probing methods → Linear Probing, Quadratic Probing, Double Hashing
  - It is used when memory is tight and load factor is low in comparision to chaining.

Page No.
Date

Coupling —

- It is the degree of dependency between classes or modules.
- Tight coupling → classes are heavily dependent on each other, harder to change / test
- Loose coupling → classes interact through interfaces or abstractions, preferred for maintainability.
- We reduce coupling by using design patterns (like Dependency injection, Observer); and by programming to interfaces instead of concrete classes.

Exception Handling —

- It is a mechanism to handle runtime errors, so normal program flow can continue.
- It uses try, catch, finally, throw and throws.
- Checked → checked at compile time, must be handled
  Unchecked → occur at runtime, not forced to handle
- Eg. IOException (checked), NullPointerException (unchecked)

⇒

```
try {
    int x = 10/0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero!");
} finally {
```

System.out.println("Finally block always executes");
}

⇒

```
public class ThrowDemo {
    static void checkAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age
            must be 18 or above");
        } else {
            System.out.println("You are eligible!");
        }
    }
    public static void main(String[] args) {
        checkAge(15);          // throws exception
    }
}
```

⇒

```
class ThrowsDemo {
    void readFile() throws IOException {
        FileReader fr = new FileReader("test.txt");
        // may throw exception
    }
    public static void main(String[] args){
        ThrowsDemo obj = new ThrowsDemo();
        try {
            obj.readFile();
        } catch (IOException e) {
            System.out.println("File not found");
        }
    }
}
```

Page No.
Date

Garbage Collection —

- It is Java's automatic memory management process that reclaims memory by destroying unused objects.
- It runs in the background and prevents memory leaks.
- In java → call "System.gc()", but JVM decides when to actually run it.
- finalize() method is called before GC deletes an object.

Wrapper class —

- In Java it converts primitive data types into objects.
- For example: int → Integer, char → Character.
- This allows primitives to be used where objects are required (like in Collections).
- Autoboxing → automatic conversion from primitive to wrapper
  int → Integer
- Unboxing → conversion from wrapper to primitive
  Integer → int

Cohesion : focus within a class (high cohesion → better)
Coupling : dependency between classes (low coupling → better)

Page No.
Date

Access Modifiers —

It controls the visibility and accessibility of classes, fields, methods and constructors within a program.

~~1) private → accessible only on~~

1) private → accessible only within the same class, only inner class can be private

⇒ class MyClass {
    private int privateField;
    private void privateMethod() {
    }
} // both accessible only within MyClass

2) default → accessible only within the same package, Package-private, no keyword

⇒ class MyClass { // In packag com
    int defaultField;
    void defaultMethod() {
    } // both accessible only with com
}

3) protected → accessible within the same package and subclasses, even if in different packages

⇒ class Parent {
    protected String protectedField;
    protected void show() {
        System.out.println("Protected method");

```
    }
}
class Child extends Parent {
    void accessProtected() {
        System.out.println(protectedField);
    }                                    // allowed
    public void display() {
        show();
    }                                    // allowed
}
```

(4) public → accessible from anywhere, including other classes and packages

⇒
```
class AnotherClass {
    public String publicField;
    public void publicMethod() {
    }
}                                        // accessible from anywhere
```

Exception Handling Keywords —

- try → block that contains code which may throw an exception
- throw → used to explicitly throw an exception
- catch → block to handle exceptions thrown in try
- finally → block that always executes (used for cleanup), whether exception occurs or not.

Page No.
Date

- throws → declares exceptions that a method might throw, so the caller must handle them.

Keywords —

- this → current object (refers to)
- super → calls parent class methods or constructors
- static → belongs to class (data → properties within class), not to objects
  shared across all instances (objects)
- final → used with :
  - variable : makes it constant
  - method : prevents overridding
  - class : prevents inheritance
- main → entry point of a Java program
- void → specifies that a method does not return any value.
- finalize → method called by GC before destroying an object (rarely used)

Method hiding : When a static method in subclass has same signature as parent's static method

Object-Oriented Programming (OOP) :
It is a programming paradigm using objects that combine data and behavior.
4 pillars: → Encapsulation, Abstraction, Inheritance, Polymorphism

```
class Student{
    String name;
    static String school;
    public static void changeschool(){
        school = "new school";
    }
}
public class CORS{
    public static void main(String args[]){
        Student.school = "ABC";
        Student stu1 = new Student();
        stu1.name = "tony";
        S.O.P(stu1.school);
    }
}
```

Money for class level

// static is used to keep data of something common for all
//directly accessed using class name

Page No.
Date

Food Delivery System

Abstract class

```
abstract class User {
    protected String name;
    protected String phone;
    public User(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }
    public abstract void displayInfo();
}
```

Inheritance

```
class Customer extends User {
    public Customer(String name, String phone) {
        super(name, phone);
    }
    @Override
    public void displayInfo() {
        System.out.println("Customer: " + name);
    }
}
class DeliveryPartner extends User {
    public DeliveryPartner(String name, String phone) {
        super(name, phone);
    }
    @Override
    public void displayInfo() {
        System.out.println("Delivery Partner: " + name);
    }
}
```

Page No.
Date

Abstract class User used because a generic user cannot exist on its own. Only a specific types like Customer or Delivery Partner should be created.
The abstract method displayInfo() ensures all child classes must implement their own version.
super() is used inside the Customer and Delivery Partner constructor to reuse the initialization logic from the parent class constructor.

Interface

```
interface PaymentMethod {
    void pay(double amount);
}
```

Polymorphism

```
class CardPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Paid with Card.");
    }
}
class UpiPayment implement PaymentMethod {
    public void pay(double amount) {
        System.out.println("Paid with UPI.");
    }
}
```

Interface PaymentMethod is used because payment types like UPI, Card are unrelated to each other but all must implement a pay() method.