Week 3
Sessions 1 & 2

9/10/2025

9/12/2025

Ch1. Overview of topics:

1. Class
2. Object
3. Abstraction / Encapsulation
4. Constructor
5. Constructor Overloading
6. Method
7. Method Overloading
8. Mutators / Accessors
9. Access Modifiers
10. ‘this’ keyword
11. References
12. Static
13. Array List
14. Packages

Class

A class is like a blueprint. An architect's plan for building a house. It describes what the house would have (rooms, doors) and what it can do (open door, turn on lights). But it's not a house yet.

```
class Car {
    String color;   // field
    int year;       // field

    void drive() {  // method
        System.out.println(x:"The car is driving.");
    }
}
```

Object

- An object is like the real house that is built from the blueprint. You can walk inside, paint it, live in it. Each object has its own unique values.

```
Car myCar = new Car();   // an object (instance)
myCar.color = "Red";
myCar.year = 2020;
myCar.drive(); // calls method
```

```java
public class BankAccount {
    private double balance; // hidden (encapsulation)

    public BankAccount(double initial) {
        balance = initial;
    }

    // Public method (abstraction)
    public void deposit(double amount) {
        balance += amount;
    }

    public double getBalance() { // Accessor
        return balance;
    }
}
```

Abstraction & Encapsulation

- While driving a car, you just use the steering wheel, pedals, and the gear shifter, but you don’t need to know the engine details.

```java
class Phone {
    String brand;
    String model;

    // Constructor
    public Phone(String b, String m) {
        brand = b;
        model = m;
    }
}

Phone p1 = new Phone(b:"Apple", m:"iPhone 15");  // constructor runs here
```

Constructor

- The constructor is a special method:
- It has the same name as the class.
- It has no return type
- Runs automatically when an object is created with the ‘new’ keyword.

Constructor overloading

- When you buy that phone, you can either:

  Take the default version ("just give me whatever is in stock").

  Customize it ("I want the iPhone 15 in blue with 256 GB").

```
class Phone {
    String brand;
    String model;

    // Constructor 1: default
    public Phone() {
        brand = "Unknown";
        model = "Generic";
    }

    // Constructor 2: custom
    public Phone(String b, String m) {
        brand = b;
        model = m;
    }
}

Phone p1 = new Phone(); // default phone
Phone p2 = new Phone(b:"Apple", m:"iPhone 15"); // customized phone
```

Method

- An action your object can perform. Like ”drive” for a car, or ”bark” for a dog.

```
class Dog {
    void bark() {
        System.out.println(x:"Woof!");
    }
}

Dog d = new Dog();
d.bark();   // makes the dog bark
```

Method overloading

- Same action as methods, but different details.
- It's like calling a pizza place. Sometimes you just say the size, other times you say size + topping.

```
class PizzaOrder {
    public void orderPizza(String size) {
        System.out.println("Ordering a " + size + " pizza.");
    }

    public void orderPizza(String size, String topping) {
        System.out.println("Ordering a " + size + " pizza with " + topping + ".");
    }
}
```

Mutators (setters)

- A setter is like a remote control. You can use it to change something inside the object, like turning the volume up on a TV. Or setting a color to be black on a car.

```
class Car {
    String color;

    public void setColor(String c) {
        color = c;
    }
}

myCar.setColor("Black");  // change the color
```

Accessors (Getters)

- Getters are like the dashboard. They let you read information without changing it. Like checking your car's speedometer. Or Checking what color your car is

```
class Car {
    String color;

    public String getColor() {
        return color;
    }
}

System.out.println(myCar.getColor());
```

Access Modifiers

```java
public class Student {
    private String name;   // private: hidden
    public int id;         // public: accessible everywhere
    protected double gpa;  // protected: accessible in subclasses

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

    public String getName() {
        return name;
    }
}
```

- Public → visible everywhere
- Private → visible only inside the class
- Protected → visible in subclasses and package
- Default (no keyword) → visible only in same package

The ‘this’ Keyword

- It’s like saying “I am Joe”/”I myself”. It points to the current object’s own fields.

```java
class Car {
    String color;

    public Car(String color) {
        this.color = color; // this.color = field, color = parameter
    }
}
```

```java
public void demoTime() {
    Time t1 = new Time();  // t1 is a reference
    Time t2 = t1;          // t2 points to the same object

    t1.setHour(hour:5);
    System.out.println(t2.getHour()); // prints 5
}
public class Time {
    private int hour;

    public void setHour(int hour) {
       this.hour = hour;
    }

public int getHour() {
    return hour;
}
```

Reference

- When you create objects with the ‘new’ keyword, you’re working with references.
- A reference stores the address of an object in memory

```java
class Car {
    static int totalCars = 0;
    public Car() {
        totalCars++;
    }
}
Car c1 = new Car();
Car c2 = new Car();
System.out.println(Car.totalCars); // prints 2
```

Static (Fields & Methods)

- A shared bulletin board in the dorm buildings. Everyone in the building (all objects of the class) can see and update it. Or increasing the number of cars needed for a Car company you work at.

```java
import java.util.ArrayList;

public class StudentList {
    Run | Debug
    public static void main(String[] args) {
        ArrayList<String> students = new ArrayList<>();

        students.add(e:"Alice");
        students.add(e:"Bob");
        students.add(e:"Charlie");

        System.out.println("First student: " + students.get(index:0));
        System.out.println("Total students: " + students.size());
    }
}
```

Array Lists

- An Array List is the first real data structure, it connects objects + references + memory.

- An array list is a resizable array that stores object references.

Packages

```java
// The Teacher class is part of the "school" package
// and has a private field "subject".

package school;   // groups classes into "school"

public class Teacher {
    private String subject;

    public Teacher(String subject) {
        this.subject = subject;
    }
}
```

- Packages are about organizing many classes/structures so they flow well after showing multiple objects in an Array List.
- They are a way to organize classes into folders.
- Example: java.util.* contains Array List, Scanner, etc.

Week 4
Sessions 3 & 4

9/17/2025

9/19/2025

Chapter 2 Inheritance

- Basics & Analogy
- Types of Inheritance
- Access Specifiers in Inheritance
- Method Overriding
- Polymorphism
- ‘super’ keyword

Inheritance

- The idea behind inheritance:
- One class can inherit fields + methods from another class.
- Think of it as a family tree: Children automatically get traits from parents, but can also add their own

Inheritance analogy

- Parent class (superclass): Animal -> all animals eat and sleep
- Child class (subclass): Dog -> dogs eat, sleep, and bark.

Inheritance code

Output:
This animal eats food.
This animal sleeps.
The dog barks.

```java
// Superclass (parent)
class Animal {
    public void eat() {
        System.out.println(x:"This animal eats food.");
    }
    public void sleep() {
        System.out.println(x:"This animal sleeps.");
    }
}

// Subclass (child)
class Dog extends Animal {
    public void bark() {
        System.out.println(x:"The dog barks.");
    }
}

public class TestInheritance {
    Run | Debug
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // inherited from Animal
        d.sleep(); // inherited from Animal
        d.bark();  // Dog's own method
    }
}
```

Types of Inheritances:

- Single inheritance
- Multilevel inheritance
- Hierarchical inheritance.

```java
class Animal {
    void eat() { System.out.println(x:"This animal eats food."); }
}

class Dog extends Animal {
    void bark() { System.out.println(x:"The dog barks."); }
}

Dog d = new Dog();
d.eat();  // from Animal
d.bark(); // from Dog
```

Single inheritance

- One child inherits from **one parent**.
- Analogy: You inherit traits from your mom.
- Another analogy is Dogs inherit traits from the class Animal.

```java
class Animal {
    void eat() { System.out.println(x:"This animal eats."); }
}

class Mammal extends Animal {
    void breathe() { System.out.println(x:"Mammal breathes air."); }
}

class Dog extends Mammal {
    void bark() { System.out.println(x:"The dog barks."); }
}

Dog d = new Dog();
d.eat();     // from Animal
d.breathe(); // from Mammal
d.bark();    // from Dog
```

Multilevel Inheritance

- A class inherits from a child class, forming a chain.
- Analogy: Traits pass from grandparent -> parent -> child.

Hierarchical Inheritance

- Multiple classes inherit from the same parent.
- Analogy: Siblings inherit traits from the same parent.

```
class Animal {
    void eat() { System.out.println(x:"This animal eats."); }
}

class Dog extends Animal {
    void bark() { System.out.println(x:"The dog barks."); }
}

class Cat extends Animal {
    void meow() { System.out.println(x:"The cat meows."); }
}

Dog d = new Dog();
d.eat();  // from Animal
d.bark();

Cat c = new Cat();
c.eat();  // from Animal
c.meow();
```

```java
class A {
    private int x = 10;
    protected int y = 20;
    public int z = 30;
}

class B extends A {
    void printValues() {
        // System.out.println(x); // ❌ private not accessible
        System.out.println(y);    // ✅ protected works
        System.out.println(z);    // ✅ public works
    }
}
```

Access Specifiers in Inheritance

- Public ➡ Everyone knows
- Protected ➡ only your family & close relatives know.
- Private ➡ only you know

Method Overriding

- The subclass redefines a method from superclass
- Same name + same parameters.
- Lets subclass provide its own version.
- Ex: The parent cooks spaghetti plain.
- The child cooks spaghetti with spices

```java
class Animal {
    void sound() { System.out.println(x:"Animal makes sound"); }
}

class Dog extends Animal {
    @Override
    void sound() { System.out.println(x:"Dog barks"); }
}

public class Test {
    Run | Debug
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Output: Dog barks
    }
}
```

```java
public class Test {
    Run | Debug
    public static void main(String[] args) {
        Animal a = new Dog();  // Reference type = Animal
        a.sound();             // Output: Dog barks
    }
}
```

Polymorphism (multiple forms)

- Polymorphism means “many forms”. Code wise, it means one interface, different behaviors.
- Compile time polymorphism -> Method Overloading
- Runtime Polymorphism -> Method Overriding
- The same word “run” -> run a race, run a business, run a program.

```java
class Parent {
    void greet() { System.out.println(x:"Hello from parent"); }
}

class Child extends Parent {
    void greet() {
        super.greet(); // call parent version
        System.out.println(x:"Hello from child");
    }
}
```

Outputs:
Hello from parent.
Hello from child.

The ‘super’ keyword

- Calls parent class constructor/method
- Lets child class use parent’s version before adding its own.
- Example: A child says, “Let my parent talk first, then I’ll add my part”.

```java
class Student {
    String name;
    Student(String name) { this.name = name; }

    @Override
    public String toString() {
        return "Student: " + name;
    }
}

public class Test {
    Run | Debug
    public static void main(String[] args) {
        Student s1 = new Student(name:"Alice");
        Student s2 = new Student(name:"Alice");

        System.out.println(s1);             // Student: Alice
        System.out.println(s1.equals(s2)); // false (different objects)
    }
}
```

Object Class

- Every class in Java extends Object automatically
- Common methods:
  toString() -> text description
  Equals() -> compares objects.
- You don't need to import it, it comes as is in Java.

Week 5
Sessions 5 & 6

9/24/2025

9/26/2025

Chapter 3 – Abstract Classes

- Abstract classes
- Abstract Methods
- Interfaces

Abstract class

- Analogy: A job description. You can't hire a "Job description," only a worker who fulfills it.
- Rules:

  - Declared with abstract.

  - Cannot be instantiated directly.

  - Can have abstract methods (no body) and concrete methods (with body).

Abstract Method

- An example is a blank recipe card. The title is there (Method name), but the steps (Method body) must be filled in by subclasses
- Rules:

  -Declared with abstract keyword

  - No body { }.

  - Subclass must provide implementation.

```
abstract class Animal {
    abstract void sound();
}

class Dog extends Animal {
    void sound() { System.out.println(x:"Woof!");
}
```

Concrete vs Abstract

- Concrete class: Can be instantiated. All methods have bodies.
- Abstract class: Cannot be instantiated. May contain abstract methods.

- A way to remember this is:

  Abstract = “template only.”

  Concrete = “finished product.”

Interfaces

```java
interface Animal {
    void sound();
    void eat();
}

class Dog implements Animal {
    public void sound() { System.out.println(x:"Woof!"); }
    public void eat() { System.out.println(x:"Dog eats kibble."); }
}
```

- An interface is like a contract. It lists what must be done, but not how.
- Rules:

  Declared with interface

  All methods are abstract (Java 7), or can be default/static (Java 8+)

  A class can implement multiple interfaces.

Abstract vs. Interface

- Abstract class:

  Can have fields + concrete methods

  Single inheritance

- Interface

  No instance fields (constants only)

  Multiple inheritance (class can implement many).

- A way to remember this:
  - Abstract class
    - Job description + Shared instructions.
  - Interface
    - Pure contract (no details).

Exit ticket

1. Why can’t you create an object of an abstract class?

2. What must a subclass do if its parent has abstract methods?

3. What’s one big difference between an abstract class and an interface?

Exit ticket Answers

1. Because an abstract class may have abstract methods (incomplete methods). You can't instantiate something that isn't fully defined.
2. The subclass must override and provide implementations for all abstract methods. Unless the subclass is also declared abstract.
3. Abstract & Interface:
    1. Abstract Classes: Can have fields and both abstract + concrete methods, and you can only extend **one** class.
    2. Interface: Only declares methods (no instance fields), but a class can implement multiple interfaces