Encapsulation Definition & Example:

Encapsulation means wrapping data (variables) and methods into a single unit called a class.
It protects the data by making variables **private** and providing **public getters and setters** to access and update them.

Why Encapsulation?

- To protect data from unauthorized access.
- To control how data is accessed or changed.
- To increase security and maintainability.

Real-time Example: Mobile Phone

- You can call, message, and use apps (methods)
- But you can't directly access or change the internal hardware parts (data)
- Or you don’t see the internal code, battery logic.

How we achieve:

- Make variables `private`
- Use public methods (getters and setters) to access and update data.

```
class Student {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

```
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if(age >= 0) {
            this.age = age;
        }
    }
}

class Main {
    public static void main(String[] args) {
        Student student = new Student();
        student.setName("Ravi");  // variable is private
        student.setAge(21);
        System.out.println(student.getName());
        System.out.println(student.getAge());
    }
}
```

Data Hiding:

Data hiding means restricting direct access to the internal variables (data) of a class.

Access Modifiers:

Access modifiers control visibility of class members:

- `private`: Access inside the same class only.
  Ex: `private int a;` – used for data hiding

- `default`: No keyword – accessible within the same package only.
  Ex: `int b;` – used when no modifier is written

- `protected`: Access within the same package and to subclasses in other packages.
  Ex: `protected int c;` – mainly used in inheritance

- `public`: Accessible from anywhere.
  Ex: `public int d;` – full access

Getters and Setters in Java:

- **Getters**: Methods used to read private variables

- **Setters**: Methods used to update private variables

Inheritance

Inheritance allows a class (child) to use properties and methods from another class (parent).

Purpose of Inheritance:

- **Code reusability** – no need to rewrite logic

- **IS-A relationship** – one class is a type of another

```
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("barking...");
    }
}
```

Dog extends Animal → Dog is-an Animal (IS-A relationship)

Types of Inheritance:

1. Single Inheritance :when a class inherits properties and methods from one parent class.

```
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("barking...");
    }
}
```

2. Multilevel Inheritance :when a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

```
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("barking...");
    }
}

class Puppy extends Dog {
    void weep() {
        System.out.println("weeping...");
    }
}
```

3. Hierarchical Inheritance : when multiple child classes inherit from a single parent class

```
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("barking...");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("meowing...");
    }
}
```

---

super Keyword in Java

The `super` keyword refers to the immediate parent class.

Uses:

1 .Call parent class method

2. Access parent class variable

3 .Call parent class constructor

```
1.class Parent {

    void show() {
```

```
        System.out.println("parent show");
    }
}

class Child extends Parent {
    void show() {
        super.show(); // calls parent's show()
        System.out.println("child show");
    }
}
```

2.
```
class Parent {
    int a = 10;
}

class Child extends Parent {
    int a = 20;
    void print() {
        System.out.println(super.a); // prints 10 (parent)
    }
}
```

3.
```
class Parent {
    Parent() {
        System.out.println("parent constructor");
    }
}

class Child extends Parent {
    Child() {
        super(); // calls parent constructor
        System.out.println("child constructor");
    }
}
```

---

extends Keyword

Used to create inheritance
→ `class B extends A` = B is a subclass of A

Method Overriding

Allows a child class to give its own specific implementation for a method that is already defined in it's superclass.

Why we use:

- To provide specific implementation in child class
- To achieve **runtime polymorphism**

Rules:

- Same method name, parameters, and return type
- Must be in parent-child relationship
- Can't override private, static, final methods
- Child method must have same or **more accessible** modifier

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

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

Ambiguity in Java

Ambiguity = confusion for the compiler
If two parents have same method, child class gets confused which one to call.
That's why Java doesn't support **multiple inheritance with classes**.

Polymorphism

Polymorphism = let us perform a single action in different ways.

Real-world Example:

In class – behave like student
In market – behave like customer

Types:

1. Compile-time Polymorphism (Method Overloading)

```
class Calculator {
    void add(int a, int b) {
        System.out.println(a + b);
    }

    void add(int a, int b, int c) {
        System.out.println(a + b + c);
    }

    void add(double a, double b) {
        System.out.println(a + b);
    }
}
```

2. Runtime Polymorphism (Method Overriding)

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

```
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}

class Main {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound();  // Dog barks
    }
}
```

Method Overloading vs Overriding

Feature	Overloading	Overriding
Class	Same class	Parent-child class
Method Name	Same	Same
Parameters	Different	Same
Return Type	Can be different	Must be same or compatible
Binding Time	Compile-time	Runtime

Abstraction

Abstraction = hiding internal details and showing only the required features
Focus on **what** an object does, not **how** it does it

Why we use:

- To reduce complexity
- To increase security

- To provide a clear interface

How to Achieve Abstraction:

1. Abstract Class

```
abstract class Animal {
    abstract void sound();  // abstract method

    void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
```

2. Interface

```
interface Vehicle {
    void start();  // abstract by default
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car started");
    }
}
```