4 PILLARS OF OOP(Object-Oriented Programming)

PILLAR 1: ENCAPSULATION

1. Introduction

In this note, I'll begin with the **first pillar of Object-Oriented Programming**, which is **Encapsulation**. Encapsulation is all about **bundling data and the methods that operate on that data into a single unit** — usually a class — and **restricting direct access to some components**. It's one of the most fundamental ways to ensure **data security**, **controlled access**, and **maintainability** in software design.

---

2. What Is Encapsulation?

Encapsulation in simple terms means **"putting data and code together and hiding the internal details from the outside world."**
It's implemented in Java using **access modifiers** like `private`, `public`, and `protected`. When we make class fields `private`, they cannot be accessed directly from outside. Instead, we provide **public methods** called **getters and setters** to control how external code can read or modify those fields.

This makes our program more **robust and secure**, because we can **validate inputs**, **restrict unauthorized access**, and prevent the internal state of an object from being changed randomly.

---

3. Real-Life Analogy

Imagine encapsulation as a **capsule of medicine**. The medicine inside is the data, and the capsule is the protection layer. The user doesn't need to know what's inside or how it's made — they only interact with the capsule safely.

In the same way, a class hides its internal logic and only exposes limited access to the outside through public methods. So just like we trust the capsule to deliver the medicine without opening it, encapsulation helps us trust that the object behaves correctly **without exposing or tampering with its internals**.

---

4. Java Implementation

Encapsulation is enforced in Java by making fields `private` and accessing them through `public` getter and setter methods.

Here’s a basic example:

```java
public class User {
    private String username;
    private String email;

    public User(String username, String email) {
        this.username = username;
        this.email = email;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        if (username != null && !username.isEmpty()) {
            this.username = username;
        }
    }
}
```

Let me explain this code step by step:

The `username` and `email` fields are declared **private**, which means no other class can access them directly.
Instead, I've provided **public methods** — like `getUsername()` and `setUsername()` — to allow controlled access.
The setter method also includes a **validation check** to make sure the username isn't empty or null. This is one of the biggest benefits of encapsulation — **we control how the field is changed**.

Now let's take this principle into my actual project — the Flight Booking System — and see how I used encapsulation in real classes like `Flight` and `User`.

---

5. Encapsulation in My Flight Booking System

In my system, encapsulation was used in **all core classes**. Let's start with the `Flight` class:

```
public class Flight {
    private String flightNumber;
    private String origin;
    private String destination;
    private int availableSeats;

    public Flight(String flightNumber, String origin, String destination, int availableSeats) {
        this.flightNumber = flightNumber;
        this.origin = origin;
        this.destination = destination;
        this.availableSeats = availableSeats;
    }

    public String getFlightNumber() {
        return flightNumber;
    }

    public int getAvailableSeats() {
        return availableSeats;
    }

    public void setAvailableSeats(int seats) {
        if (seats >= 0) {
            this.availableSeats = seats;
        }
    }
}
```

Now explaining this:

All data fields like flight number, origin, destination, and seat count are **kept private**.
This means they can't be accessed directly from other classes like `Passenger` or `Admin`.
Instead, I provide **getter methods** to read values, and **setter methods** with conditions — for example, I only allow setting available seats if the number is non-negative.

This way, I **prevent invalid or dangerous data from entering the system**, like a negative number of seats. Encapsulation ensures that the internal state of every object stays **safe, clean, and predictable**.

---

6. Design Benefits of Encapsulation

Now let's move on to why encapsulation is not just a good practice, but a **necessary design principle**:

- **Security**: By hiding internal data, we prevent accidental or malicious modification.

- **Maintainability**: If we change the internal logic, the rest of the system is unaffected as long as public methods remain the same.
- **Flexibility**: We can change how a field is calculated or stored without changing how other classes use it.
- **Modularity**: Each class acts as a self-contained unit with a clear interface, improving testing and debugging.

---

7. Final Summary

Encapsulation is about **hiding the internal state of objects** and only exposing a controlled interface. It's implemented using `private` fields and `public` getter and setter methods.

In my Flight Booking System, I applied encapsulation in classes like `User`, `Flight`, and `Passenger` to enforce security and data consistency.
This allowed me to build a system where each object manages its own data responsibly, and the rest of the code only interacts with that data in safe, predictable ways.

PILLAR 2: ABSTRACTION

1. Introduction

In this note, I will be explaining the **second pillar of Object-Oriented Programming**, which is **Abstraction**. Abstraction is one of the most powerful OOP concepts because it allows us to **focus on what an object does, rather than how it does it**.

In my project — the Flight Booking System — I used abstraction in several places to design a clean, modular structure where each part of the program exposes only the essential details and hides the internal logic.

---

2. What is Abstraction?

Abstraction means **hiding the complexity** of implementation and **showing only the necessary details**.
In other words, it allows the programmer to create a simplified interface for using objects — while keeping the internal working hidden from the user.

In Java, we achieve abstraction using either **abstract classes** or **interfaces**.
With **abstract classes**, we can define both abstract methods (which don't have a body) and concrete methods (which do).
With **interfaces**, we can define a contract — a list of methods that any implementing class must provide.

---

3. Real-Life Analogy

Think of abstraction like **driving a car**. When you're driving, you just use the **steering wheel, brake, and accelerator**. You don't need to know how the fuel injectors work or how the transmission transfers power.

That's abstraction — the car provides a simple interface for the driver, while hiding all the internal mechanics.
In programming, this means we can **interact with objects** through well-defined methods, without needing to understand their internal complexity.

---

4. Java Implementation

Let’s look at a simple abstract class in Java:

```
public abstract class User {
    protected String username;
    protected String email;

    public User(String username, String email) {
        this.username = username;
        this.email = email;
    }

    public abstract void displayRole();

    public void login() {
        System.out.println("User logged in.");
    }
}
```

Now explaining this:

In this example, I’ve declared `User` as an **abstract class** — meaning it can’t be instantiated directly.
Inside it, I’ve defined one abstract method called `displayRole()`. This method has **no body** — it’s a placeholder, and each subclass must provide its own implementation.
I’ve also added a **concrete method** called `login()` which can be reused by all subclasses. So this class partially defines functionality, and leaves some behavior to be defined by child classes.

This gives us a **blueprint**. Now let’s look at how subclasses use this.

```java
public class Passenger extends User {
    public Passenger(String username, String email) {
        super(username, email);
    }

    @Override
    public void displayRole() {
        System.out.println("Role: Passenger");
    }
}
```

And here's another one:

```java
public class Admin extends User {
    public Admin(String username, String email) {
        super(username, email);
    }

    @Override
    public void displayRole() {
        System.out.println("Role: Admin");
    }
}
```

Now explaining this part:

Both `Passenger` and `Admin` extend the abstract class `User`, and they each **override** the `displayRole()` method with their own specific message.
At the same time, they **reuse the `login()` method** from the abstract class without needing to write it again.
This is the power of abstraction — we define what the system needs to do at a high level, and each class can customize how it behaves.

---

5. Abstraction in My Flight Booking System

In my Flight Booking System project, I used abstraction to define **general behaviors** in the `User` class, and allow each subclass — like `Admin` and `Passenger` — to **implement their own roles** while following a common structure.

This made the code **modular and scalable**. If I wanted to add a new role tomorrow — for example, a `Staff` or `FlightManager` — I would simply extend the abstract `User` class and implement the missing abstract methods.

Abstraction also helped in organizing flight-related logic. For example, I could have defined an interface like:

```
public interface Bookable {
    void bookFlight(Flight flight);
}
```

Then my `Passenger` class would implement this:

```
public class Passenger extends User implements Bookable {
    public void bookFlight(Flight flight) {
        // booking logic
    }
}
```

This way, abstraction lets us define **common contracts** like `Bookable`, and any class that wants to book flights must implement that method.

---

6. Benefits of Abstraction

Now let's talk about the design benefits of abstraction:

- **Simplified interfaces** – users of the class don't need to understand the full implementation
- **Security** – sensitive internal logic is hidden from external classes
- **Reusability** – abstract classes allow shared methods and structures

- **Extensibility** – you can add new subclasses easily without changing existing code
- **Focus on behavior** – abstraction lets developers think in terms of actions rather than internal details

Abstraction is especially useful in larger projects like mine, where different user types perform different roles but follow the same base behavior.

---

7. Final Summary

To summarize:
Abstraction helps us design systems that are **modular, secure, and easy to extend**. We achieve this in Java using **abstract classes and interfaces**.
In my Flight Booking System, I created an abstract `User` class that defines shared behavior, and subclasses like `Passenger` and `Admin` that each implement role-specific features.

PILLAR 3: POLYMORPHISM

1. Introduction

In this note, I'll now explain the **third pillar of Object-Oriented Programming**, which is **Polymorphism**. The term polymorphism comes from Greek — *poly* means many, and *morph* means forms — so polymorphism literally means **"many forms."**

In object-oriented programming, **polymorphism allows a single interface to behave differently depending on the object that's using it**. This makes our code more flexible, extensible, and maintainable — especially when dealing with classes like `User`, `Admin`, and `Passenger` in a large system like my Flight Booking System.

---

2. What Is Polymorphism?

There are **two types of polymorphism** in Java:

1. **Compile-time polymorphism** – also called **method overloading**
2. **Runtime polymorphism** – also known as **method overriding**

Let me break these down:

- **Method Overloading** means having multiple methods with the same name in the same class, but with different parameter types or numbers. The correct method is chosen **at compile-time** based on the argument types.
- **Method Overriding**, on the other hand, is when a subclass provides a **specific implementation** of a method that is already defined in its superclass. The correct method is selected **at runtime** based on the object type.

Polymorphism supports **dynamic behavior**, allowing us to write more reusable and scalable code.

---

3. Real-Life Analogy

Think of the action **"drive"**.
A person can drive a car, a truck, or a motorcycle. The **interface is the same** — drive — but the actual behavior is different depending on the vehicle.

Similarly, in programming, we can call the same method — for example, `displayRole()` — on different objects like `Passenger` or `Admin`, and each of them will respond in their own unique way.
That's polymorphism — same name, different behavior.

---

4. Java Implementation – Overriding

Let's start with **runtime polymorphism** or method overriding.

Here's my abstract `User` class:

```
public abstract class User {
    protected String username;
    protected String email;

    public User(String username, String email) {
        this.username = username;
        this.email = email;
    }

    public abstract void displayRole();

    public void login() {
        System.out.println(username + " logged in.");
    }
}
```

Now, subclasses override the displayRole() method:

```java
public class Passenger extends User {
    public Passenger(String username, String email) {
        super(username, email);
    }

    @Override
    public void displayRole() {
        System.out.println("Role: Passenger");
    }
}
```

java

```java
public class Admin extends User {
    public Admin(String username, String email) {
        super(username, email);
    }

    @Override
    public void displayRole() {
        System.out.println("Role: Admin");
    }
}
```

Now here's how polymorphism works in practice:

```java
User user1 = new Passenger("ashok", "ashok@email.com");
User user2 = new Admin("dendul", "admin@email.com");

user1.displayRole(); // Output: Role: Passenger
user2.displayRole(); // Output: Role: Admin
```

Even though both objects are declared as `User`, the `displayRole()` method behaves **differently** depending on the actual object type.
This is **runtime polymorphism** — the method that gets called is chosen dynamically based on the object instance.

This helped me keep my code flexible — I could store a list of users of different types, and still call methods like `login()` or `displayRole()` without knowing the exact subclass.

---

5. Java Implementation – Overloading

Now let's move on to **method overloading**, or **compile-time polymorphism**. This means we can define multiple methods with the **same name** but with different parameters.

Here's an example in the `FlightService` class:

```
public class FlightService {

    public void searchFlight(String destination) {
        System.out.println("Searching flight to " + destination);
    }

    public void searchFlight(String origin, String destination) {
        System.out.println("Searching flights from " + origin + " to " + destination);
    }

    public void searchFlight(String origin, String destination, String date) {
        System.out.println("Searching flights from " + origin + " to " + destination + " on " + date);
    }
}
```

Here we have three methods with the same name `searchFlight()`, but each has a different number of parameters.

When I call `searchFlight("Kathmandu")`, the first method is used.
If I call `searchFlight("Pokhara", "Kathmandu")`, the second one is triggered.
And if I add a date, the third version is used.

This is polymorphism at **compile-time** — the compiler selects the method based on arguments.

This allowed me to give **multiple search options** in the booking system while keeping the method name consistent for the user.

---

6. Polymorphism in My Flight Booking System

In my project, polymorphism allowed me to treat all users under a **single reference type** — `User` — and then rely on Java to handle the behavior dynamically.

For example:

```
List<User> userList = new ArrayList<>();
userList.add(new Passenger("Ram", "ram@email.com"));
userList.add(new Admin("Shyam", "shyam@email.com"));

for (User u : userList) {
    u.displayRole(); // Each user shows their role properly
}
```

Even though I'm looping through `User` objects, Java calls the correct version of `displayRole()` based on whether the object is an `Admin` or a `Passenger`. This made my code **scalable and future-proof**.

It also meant I could **extend** the system later — for example, adding a new user type like `SupportStaff` — and the same logic would work without changes. That's the power of polymorphism.

---

7. Design Benefits of Polymorphism

Polymorphism provides several major benefits:

- **Flexibility** – same interface, different implementations
- **Extensibility** – easy to add new behavior without modifying existing code
- **Cleaner code** – fewer conditional statements like `if-else`
- **Testability** – you can test classes independently

It supports both **DRY** (Don't Repeat Yourself) and **Open/Closed Principle** — meaning classes are open for extension but closed for modification.

---

8. Final Summary

To summarize:
Polymorphism means **one method name can perform different tasks**, depending on the object or parameters.
Java supports **method overloading** for compile-time polymorphism and **method overriding** for runtime polymorphism.

In my Flight Booking System, I used polymorphism to simplify user management, provide multiple ways to search flights, and handle behavior dynamically — all while keeping my code clean, scalable, and maintainable.

PILLAR 4: INHERITANCE

Introduction & Overview

In this note, I'll be explaining the **fourth pillar of Object-Oriented Programming**, which is **Inheritance**. Inheritance is a key concept in OOP that allows one class to inherit properties and methods from another. It supports **code reuse**, **logical hierarchy**, and makes it easy to extend existing functionality.

Now that I've already discussed encapsulation, abstraction, and polymorphism, let's move on to understand how **inheritance fits into the bigger picture**, especially with how I've applied it in my **Flight Booking System**.

---

2. Definition & Purpose

Inheritance is a feature of object-oriented programming that allows a class, known as a **subclass**, to inherit data members and methods from another class, known as a **superclass**.

Now, the purpose of inheritance is to promote **code reuse** and **establish hierarchical relationships** between classes. For instance, we might have a generic `User` class with shared fields like username and email. Then we could have subclasses like `Passenger`, `Admin`, and `Staff`, each of which inherit the basic properties and then define **role-specific functionality**.

This avoids redundancy, simplifies maintenance, and allows us to write flexible, extensible code.

---

3. Real-World Analogy

Imagine a class called `Person`, which has general features like name, date of birth, and methods like eat and sleep. Now from this base class, we can derive subclasses like `Student`, `Doctor`, or `Teacher`.

Each of these roles is a **specific version of a person**, and inherits all general behavior while adding new ones — like `submitAssignment()` for students or `prescribeMedicine()` for doctors.

This shows the **"is-a" relationship**, which is central to inheritance: a student *is a* person, a doctor *is a* person. Similarly, in programming, inheritance allows us to model this hierarchy in code.

---

4. Java Implementation & Types

In Java, we use the `extends` keyword to implement inheritance. Here's a basic example:

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

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

Now explaining this — the Dog class inherits from the Animal class and overrides the makeSound() method to provide more specific behavior. This not only demonstrates inheritance but also polymorphism, where the same method name behaves differently depending on the object.

Now let's discuss types of inheritance. Java supports:

Single inheritance – where one class inherits from one parent.

Multilevel inheritance – a class inherits from another which itself extends a third.

Hierarchical inheritance – multiple classes share the same parent.

Java does not support multiple inheritance of classes to avoid ambiguity from the diamond problem. Instead, Java solves this through interfaces, which offer a way to achieve multiple inheritance in a safe and structured way.

---

5. Inheritance in My Flight Booking System

In my project, I created a general class called `User`. This class includes shared properties like `username`, `email`, and methods like `login()` and `logout()`. These are common across all users of the system — whether they are Admins, Passengers, or Staff.

example,

```
public class Passenger extends User {
    private List<Flight> bookedFlights;

    public Passenger(String username, String email) {
        super(username, email);
        this.bookedFlights = new ArrayList<>();
    }

    public void bookFlight(Flight flight) {
        bookedFlights.add(flight);
        System.out.println("Flight booked: " + flight.getFlightNumber());
    }
}
```

Now explaining this code:
The `Passenger` class extends the `User` class, meaning it **inherits all user-related fields and methods**. In the constructor, I use the `super()` keyword to call the parent class constructor and initialize the inherited fields.
Then I declare a new property called `bookedFlights`, which is a list of all the flights this passenger has booked.
The method `bookFlight()` takes a `Flight` object, adds it to the list, and prints a confirmation.
This is a classic example of **extending inherited functionality** — the passenger can still log in using the parent method, but now they also have booking capabilities.

Now let's look at the `Admin` class:

```
public class Admin extends User {
    public Admin(String username, String email) {
        super(username, email);
    }

    public void addFlight(Flight flight, List<Flight> flightList) {
        flightList.add(flight);
        System.out.println("Flight added: " + flight.getFlightNumber());
    }
}
```

Here, the `Admin` class also **inherits from `User`**, gaining access to fields like `username` and methods like `login()`.
In the constructor, again, `super(username, email)` initializes inherited properties.
The `addFlight()` method allows admins to add new flights to the system's master list.
It takes a `Flight` object and a list of all available flights, adds the new flight to the list, and confirms with a print statement.

So both `Passenger` and `Admin` share common functionality through inheritance, but they specialize it based on their role. This design improves modularity and scalability — if tomorrow I add a new role, say `AirportManager`, I can simply extend the `User` class and add role-specific logic.

---

6. Design Principles & Best Practices

When applying inheritance, I've made sure to follow best practices:

- Only use inheritance for **true "is-a" relationships**, like `Passenger is a User`.
- Avoid deep inheritance hierarchies — I kept it shallow and easy to manage.
- Followed the **Liskov Substitution Principle** — all subclasses can replace the parent without breaking functionality.
- I also kept shared logic in the parent class, so updates like changing how login works only have to be made in one place.

---

7. Final Summary & Closing

To summarize:
Inheritance allows classes to share functionality, reduce redundancy, and maintain a clean, organized codebase. In Java, this is implemented using `extends`.
In my Flight Booking System, the `User` class served as a base, while `Passenger` and `Admin` extended it to include booking and management functionality, respectively.