✅ Case Study: Liskov Substitution Principle (LSP)

Domain: Banking → Account Types (Savings, FixedDeposit)

🎯 Business Context:

Rajeev is building a banking application that handles **Savings Accounts** and **Fixed Deposit Accounts**. Both accounts allow querying balance, but **only Savings Accounts** allow **withdrawals**.

He uses inheritance — creates a `FixedDepositAccount` as a subclass of `BankAccount`, but this leads to a **violation of LSP**.

❌ LSP Violation – Problematic Design

Superclass

```
public class BankAccount {
    protected double balance;

    public BankAccount(double balance) {
        this.balance = balance;
    }

    public void withdraw(double amount) {
        balance -= amount;
        System.out.println("Withdrawn: " + amount);
    }

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

Subclass: FixedDepositAccount

```
public class FixedDepositAccount extends BankAccount {

    public FixedDepositAccount(double balance) {
        super(balance);
    }

    // Override to prevent withdrawal
    @Override
    public void withdraw(double amount) {
        throw new UnsupportedOperationException("Withdrawals not allowed from
FD");
    }
}
```

Test Code – Violates LSP

```
public class BankTest {
    public static void main(String[] args) {
        BankAccount account = new FixedDepositAccount(10000);
```

```
        account.withdraw(1000); // ❌ Violates LSP – client expects it to work
    }
}
```

⚠ Why This Violates LSP?

> "Objects of a superclass should be replaceable with objects of a subclass without breaking the application."

- We replaced `BankAccount` with a `FixedDepositAccount`, but it **crashed**.
- **Contract of `withdraw()` is broken** — clients can't rely on the behavior anymore.
- This breaks polymorphism and makes the code fragile and unsafe.

✅ Refactored Code – LSP Compliant

Instead of forcing inheritance, use **interface segregation** to model account capabilities.

Step 1: Common Account Interface

```
public interface Account {
    double getBalance();
}
```

Step 2: Withdrawable Interface

```
public interface Withdrawable {
    void withdraw(double amount);
}
```

Step 3: SavingsAccount (supports withdrawal)

```
public class SavingsAccount implements Account, Withdrawable {
    private double balance;

    public SavingsAccount(double balance) {
        this.balance = balance;
    }

    @Override
    public void withdraw(double amount) {
        balance -= amount;
        System.out.println("Withdrawn: " + amount);
    }

    @Override
    public double getBalance() {
        return balance;
```

```
    }
}
```

Step 4: FixedDepositAccount (no withdrawal)

```
public class FixedDepositAccount implements Account {
    private double balance;

    public FixedDepositAccount(double balance) {
        this.balance = balance;
    }

    @Override
    public double getBalance() {
        return balance;
    }
}
```

✅ LSP-Safe Test Code

```
public class BankTest {
    public static void main(String[] args) {
        Account fd = new FixedDepositAccount(10000);
        Account sa = new SavingsAccount(5000);

        System.out.println("FD Balance: " + fd.getBalance());
        System.out.println("SA Balance before: " + sa.getBalance());

        if (sa instanceof Withdrawable) {
            ((Withdrawable) sa).withdraw(1000);
        }

        System.out.println("SA Balance after: " + sa.getBalance());
    }
}
```

🧠 Real-World Analogy

You have two lockers:

- **Savings locker**: you can withdraw money any time.
- **FD locker**: once money is locked, you can't withdraw early.

If you treat all lockers the same and expect withdrawal from FD, it will **break your trust** in the locker system.

✅ Summary of LSP Benefits

Principle	Benefit
Behavioral	You can use subclasses without worrying about different behavior

Principle	Benefit
Substitutability	
Code Safety	Prevents runtime surprises (e.g., `UnsupportedOperationException`)
Flexible Design	Models real capabilities, avoids misuse of inheritance

---

👩‍🏫 Teaching Tips

- Ask: "Can you substitute any subclass in the place of the superclass?"
- Use this case in discussions on **composition over inheritance**
- Add a twist: "Introduce another type of account (e.g., LoanAccount). Can it also fit cleanly?"
