LECTURE 1

Object-Oriented Foundations & Principles Revisited

Advanced Object-Oriented Programming

Duration: 4 Hours | LO1: Critically evaluate OOP principles

Today's Agenda

4 Hours — Building Foundations for the Semester

Hour 1

Introducing the Running Use Case: SmartShelf

A Smart Campus Library System that grows with us each week

Hour 2

The Four Pillars of OOP — Deep Dive

Encapsulation, Abstraction, Inheritance, Reusability — critically evaluated

Hour 3

Mutable vs Immutable Objects

Understanding object state, side effects, and design implications

Hour 4

Workshop: Hands-On with SmartShelf

Build the first domain classes, apply principles practically

Hour 1

Introducing SmartShelf — Our Running Use Case

SmartShelf: Smart Campus Library System

A real-world system we'll build incrementally over 9 weeks

The Vision

A university library needs a modern management system that handles:

- Books, journals, DVDs, and digital media
- Student & staff memberships
- Borrowing, returns, and reservations
- Fines, notifications, and reporting
- Integration with external catalog APIs

9-Week Roadmap

W1: OOP Foundations — Domain classes

W2: Cloning — Book copies & references

W3: Polymorphism — Media types & search

W4: Generics — Typed catalogs & LSP

W5: Exceptions — Loan validation & DbC

W6: Creational Patterns — Factories

W7: Structural Patterns — Adapters

W8: Behavioural — Observers & strategies

W9: Integration & Project Workshop

SmartShelf — Core Domain Entities

The building blocks we'll start with today

Book

Attributes

`isbn, title, author, publishYear, genre, isAvailable`

Member

Attributes

`memberId, name, email, memberType, borrowedBooks, registrationDate`

Library

Attributes

`name, catalog (list of books), members (list of members), loanRecords`

Why Object-Oriented Programming?

Procedural Approach

- Data and functions are separate
- Global state, difficult to track changes
- Hard to reuse — copy-paste culture
- Modifications ripple across codebase
- Difficult to model real-world domains

Object-Oriented Approach

- Data and behaviour bundled together
- Encapsulated state, controlled access
- Inheritance & composition for reuse
- Changes are localized to objects
- Natural mapping to real-world entities

Hour 2

The Four Pillars of OOP — A Critical Lens

The Four Pillars of OOP

LO1: Critically evaluate the utility of these principles

Encapsulation

Bundling data with methods that operate on it. Controlling access.

Abstraction

Hiding complexity. Exposing only what's necessary.

Inheritance

Creating new classes from existing ones. Hierarchical reuse.

Reusability

Writing code once, using it across different contexts.

Encapsulation

Protecting data integrity through controlled access

Key Concepts

- Private fields — hide internal state
- Public getters/setters — controlled access
- Validation logic inside setters
- Information hiding — reduce coupling
- Class as a "contract" with the outside

SmartShelf Example

```
class Book {
  private String isbn;
  private String title;
  private boolean available;

  public boolean isAvailable() {
    return this.available;
  }

  public void setIsbn(String isbn) {
    if (isbn == null || isbn.isEmpty())
      throw new IllegalArgumentException();
    this.isbn = isbn;
  }
}
```

Encapsulation — Critical Evaluation

Strengths

- Prevents invalid states — setters enforce rules
- Reduces coupling — internal changes don't break clients
- Easier to debug — state changes are traceable

Limitations & Critiques

- "Getter/setter" anti-pattern — just boilerplate?
- Over-encapsulation can make code rigid
- Reflection & serialization can bypass access control

Discussion Prompt

In SmartShelf, should Book expose setTitle() publicly? What if a cataloging system needs to correct a title after creation? How do we balance flexibility vs. data integrity?

Abstraction

Hiding complexity, exposing only essential features

Concept

- Abstract classes define "what" not "how"
- Interfaces declare capabilities
- Clients depend on abstractions, not details
- Simplifies complex systems into manageable parts
- Real-world analogy: you drive a car without knowing engine internals

SmartShelf: Searchable Interface

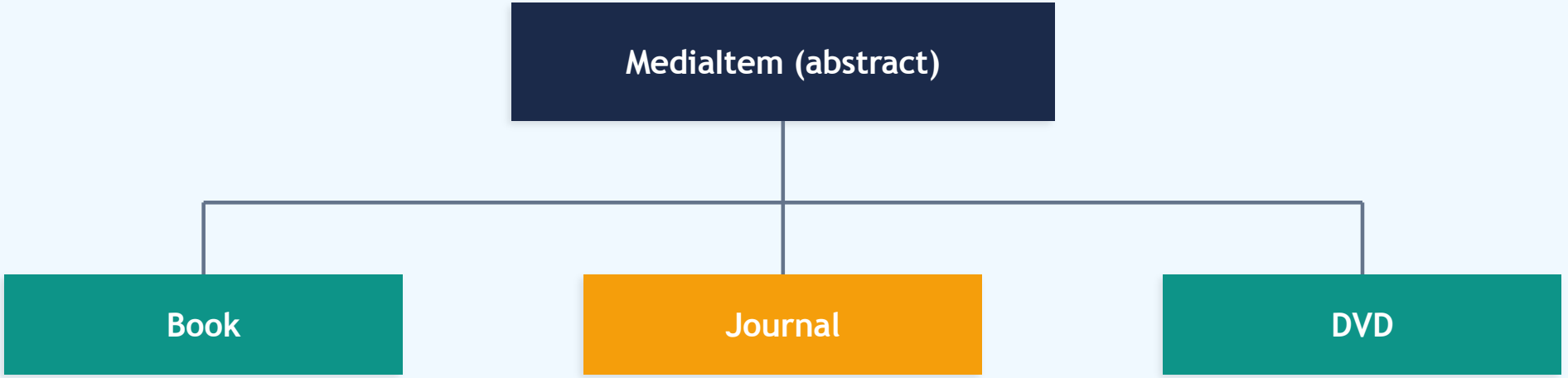
```
interface Searchable {
  List<Book> search(String query);
}

// Library uses Searchable
// without knowing the algorithm
class Library {
  private Searchable searchEngine;

  public List<Book> findBooks(String q){
    return searchEngine.search(q);
  }
}
```

Inheritance

Creating specialised types from general ones



Benefits: code reuse, natural hierarchy, polymorphic behaviour via parent type references

Caution: deep hierarchies create fragile code. Favor composition over inheritance when relationships aren't truly "is-a."

Reusability

Write once, use everywhere — but how effectively?

Inheritance

Subclass reuses parent's code. Quick but creates tight coupling.

Composition

Objects contain other objects. Flexible, loosely coupled, preferred.

Interfaces

Define contracts. Any class that implements can be swapped in.

SmartShelf: Reusable Notification interface

```
interface Notifiable { void notify(String message); }
// EmailNotifier, SMSNotifier, PushNotifier all implement Notifiable
// Library class uses Notifiable — doesn't care which implementation
```

Critical Evaluation of OOP Principles

LO1 — When do these principles help, and when do they hinder?

Principle	Utility (When It Helps)	Pitfall (When It Hurts)
Encapsulation	Prevents invalid state, isolates change	Boilerplate getters/setters, over-hiding
Abstraction	Manages complexity, clean APIs	Leaky abstractions, wrong boundaries
Inheritance	Natural hierarchies, code reuse	Fragile base class, tight coupling
Reusability	Reduces duplication, faster development	Premature generalisation, wrong abstractions

Key Takeaway: OOP principles are tools, not rules. A skilled developer knows when to apply them and when to break them intentionally.

Hour 3

Mutable & Immutable Objects

Mutable vs Immutable Objects

Understanding object state and its design consequences

Mutable Objects

- State can change after creation
- Setter methods modify internal fields
- Must handle concurrent access carefully
- Example: Member's borrowedBooks list changes as books are checked out/returned

Use when: state legitimately changes over the object's lifetime

Immutable Objects

- State cannot change after creation
- All fields are final, no setters
- Thread-safe by design
- Example: an ISBN object — once assigned, it should never change

Use when: value represents a fixed identity or measurement

Building an Immutable ISBN Class

SmartShelf — ISBN as a value object

```
public final class ISBN {
  private final String value;

  public ISBN(String value) {
    if (!isValid(value))
      throw new IllegalArgumentException(
        "Invalid ISBN: " + value);
    this.value = value;
  }

  public String getValue() {
    return this.value;
  }

  private static boolean isValid(String v){
    return v != null && v.length() == 13;
  }
}
```

Immutability Recipe

- Class declared final
- All fields private final
- No setter methods
- Validation in constructor
- Defensive copies of mutable fields (if any)

Design Decision: Mutable or Immutable?

SmartShelf Class	Mutable / Immutable?	Reasoning
ISBN	**Immutable**	Identity value — never changes once assigned
Book	**Mutable**	Availability changes as books are borrowed/returned
Member	**Mutable**	Borrowed list, contact info can update
LoanRecord	**Immutable**	Historical record — once created, shouldn't change
DateRange	**Immutable**	Value object — represents a fixed period

Principle: Default to immutable. Only make mutable what genuinely needs to change after creation.

The Hidden Dangers of Mutability

Why shared mutable state is a source of bugs

Aliasing Bug in SmartShelf

```
Book book = library.getBook("978-0...");

// Two references to the SAME object
Book ref1 = book;
Book ref2 = book;

ref1.setAvailable(false);

// Surprise! ref2 also sees the change
System.out.println(ref2.isAvailable());
// Output: false

// Both ref1 and ref2 point to
// the same mutable object in memory
```

Consequences

- Aliasing: multiple references to same mutable object
- Unexpected side effects across modules
- Race conditions in concurrent code
- Difficult to reproduce and debug

Mitigation:

- Defensive copies
- Immutable value objects
- Copy-on-write patterns

Hour 4

Hands-On Workshop: Building SmartShelf v0.1

Workshop Exercise 1: Core Domain Classes

Build the foundation of SmartShelf — 45 minutes

Task A — **Create the ISBN class (immutable)**

Final class, validated constructor, proper equals() and hashCode(), no setters

Task B — **Create the Book class (mutable)**

Private fields, public getters, controlled setters with validation, uses ISBN as a value object

Task C — **Create the Member class**

Encapsulated borrowedBooks list, methods to borrow/return books, membership validation

Checklist: Proper encapsulation? Immutability where appropriate? Validation in constructors/setters? Clean separation of concerns?

Workshop Exercise 2: Library Class & Integration

Wire the domain together — 45 minutes

Task D

Create the Library class

Manages a catalog (List<Book>), members list, and provides addBook(), registerMember() methods

Task E

Implement borrowBook(memberId, isbn)

Validate member exists, book available, enforce max borrow limit (3), update both Book and Member state

Task F

Write a Main class to demonstrate

Create library, add books, register members, borrow/return books — print state at each step

Discussion: What happens if two threads call borrowBook() for the same book simultaneously? We'll address this in Week 5.

What's Coming Next

Week 2: Cloning — Deep & Shallow Cloning

SmartShelf Evolution: Cloning Books for Reservations

When a student reserves a book, should we clone the Book object or just copy the reference?

What happens when a Book contains a mutable Author object — does a shallow copy suffice?

We'll implement Cloneable interface, explore clone() pitfalls, and build a proper deep copy mechanism.

Preparation: Read Head First Java, Chapter on Object Lifecycle. Complete Tasks A–F from today's workshop and commit to your repository.

Lecture 1 Summary

- SmartShelf use case introduced — our semester-long project
- Encapsulation: controlled access, data integrity
- Abstraction: hiding complexity behind clean interfaces
- Inheritance: hierarchical reuse (with caution)
- Reusability: composition over inheritance
- Mutable vs Immutable: default to immutable

LO1 Addressed: Critically evaluate the utility of OOP principles