Advanced Object-Oriented Programming · SmartShelf Project

LECTURE 6

SOLID Principles & Creational Patterns

Singleton · Factory · Builder · Prototype

4 Hours · Dr. T. Madushanka · University of Kelaniya

Session Overview

Lecture 6

Hour 1

SOLID Principles & DIP Deep Dive

S-O-L-I-D recap · Dependency Inversion deep dive · Before/After refactoring in SmartShelf

Hour 2

Singleton & Factory Patterns

Singleton risks & enum idiom · Factory Method · Abstract Factory · BookFactory in SmartShelf

Hour 3

Builder & Prototype Patterns

Builder for complex LoanRequest objects · Fluent API · Prototype vs new · BookTemplate cloning

Hour 4

Workshop — All 4 Patterns

Tasks A–D: DIP refactoring · LibraryRegistry singleton · MediaFactory · LoanRequestBuilder

SmartShelf Journey — Where We Are

Lecture 6

L1

Encapsulation

ISBN, Book, Member, Library

L2

Inheritance & Polymorphism

MediaItem, DVD, Magazine hierarchies

L3

Abstract Classes & Interfaces

Borrowable, Searchable, Reportable

L4

Generics & LSP

Catalog<T>, Result<T>, Pair<K,V>

L5

Exceptions & Design by Contract

Custom exceptions, Contracts, DbC

L6

SOLID + Creational Patterns

DIP, Singleton, Factory, Builder, Prototype

HOUR 1

SOLID Principles & DIP

Why dependency direction matters — and how to fix it

The 5 SOLID Principles

Lecture 6

S — **Single Responsibility**

A class should have one reason to change

SmartShelf: Library vs LibraryReporter

O — **Open / Closed**

Open for extension, closed for modification

SmartShelf: New MediaType without touching Library

L — **Liskov Substitution**

Subtypes must be substitutable for base types

SmartShelf: ReadOnlyMedia violation (covered L4)

I — **Interface Segregation**

Clients shouldn't depend on interfaces they don't use

SmartShelf: Borrowable, Searchable, Reportable

D — **Dependency Inversion**

Depend on abstractions, not concrete classes

SmartShelf: TODAY: Library ← INotifier ← EmailNotifier

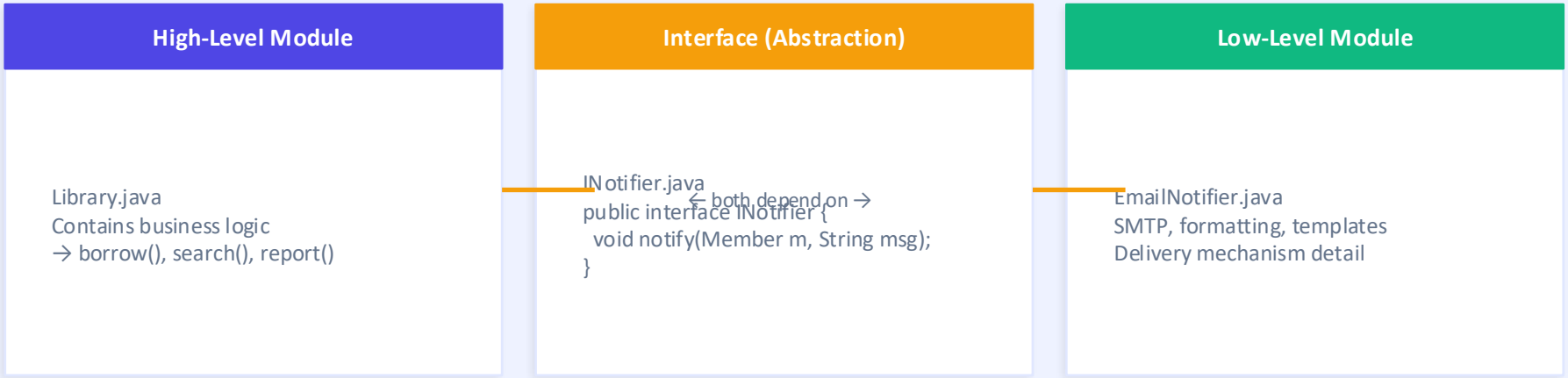
Dependency Inversion Principle — Deep Dive

Lecture 6

Two Rules:

① High-level modules should NOT depend on low-level modules. Both should depend on abstractions.

② Abstractions should NOT depend on details. Details (concrete classes) should depend on abstractions.



DIP in SmartShelf — Before & After

Lecture 6

❌ BEFORE — Tight Coupling

```
// Library directly creates EmailNotifier
public class Library {
  private EmailNotifier notifier;
  //  ↑ depends on CONCRETE class!

  public Library() {
    this.notifier = new EmailNotifier();
  }

  public void borrowBook(...) {
    // ...
    notifier.sendEmail(member, msg);
```

Problem: Cannot swap SMS notifier. Unit tests break. Library is polluted with delivery logic.

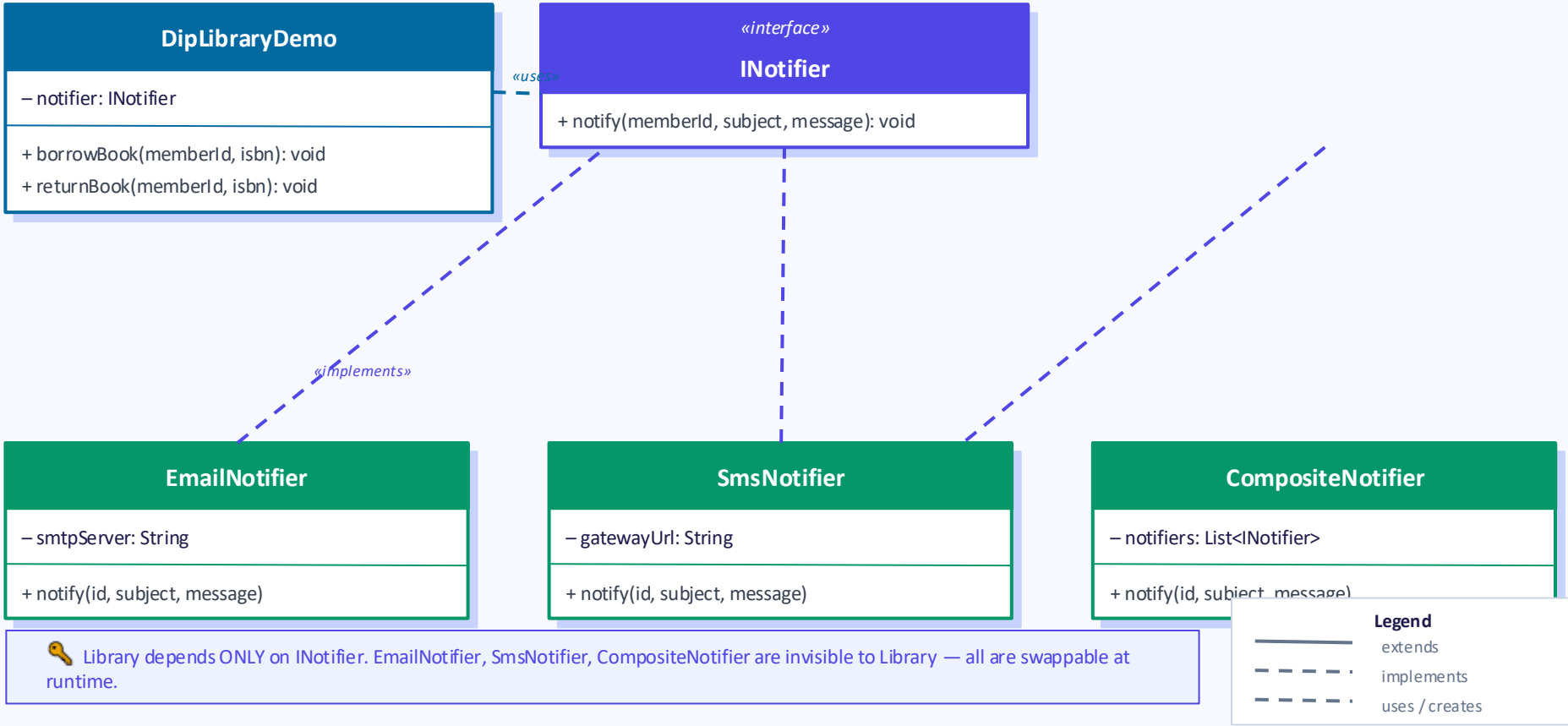
✅ AFTER — Dependency Inversion

```
// Depend on INotifier interface
public class Library {
  private final INotifier notifier;
  //  ↑ depends on ABSTRACTION

  public Library(INotifier notifier) {
    this.notifier = notifier;
  }

  public void borrowBook(...) {
    notifier.notify(member, msg);
  }
```

Benefit: Swap Email → SMS → Slack notifier. Mock in tests. Library stays clean.

🔑 Rule: Constructor injection (passing via constructor) is the preferred DIP implementation — no reflection, no frameworks needed.

Class Diagram — Dependency Inversion Principle (DIP)



HOUR 2

Singleton & Factory Patterns

Controlling instantiation — one instance or many, your way

Singleton Pattern — Intent & Implementation

Lecture 6

Guarantee exactly one instance of a class exists in the JVM.

❌ Naive (broken)

```
private static LibraryRegistry
  instance;

public static LibraryRegistry
  getInstance() {
  if (instance == null)
    instance = new
LibraryRegistry();
  return instance;
}
```

Not thread-safe!
Two threads → two instances

✅ Double-Checked

```
private static volatile
  LibraryRegistry instance;

public static LibraryRegistry
  getInstance() {
  if (instance == null) {

synchronized(LibraryRegistry.class
)
    { if(instance==null)
        instance = new
LibraryRegistry();}
  } return instance;
}
```

Thread-safe but
verbose & tricky

✅ Enum (Best)

```
public enum LibraryRegistry {
  INSTANCE;

  private final
Map<String,Library>
    registeredLibraries =
      new HashMap<>();

  public void register(
    String id, Library lib) {
    registeredLibraries.put(id,
lib);
  }
}
```

Thread-safe, serialization-
safe, concise — preferred

Singleton Trade-offs — The Honest Truth

Singleton is the most misused pattern. Know when not to use it.

✅ When Singleton Makes Sense

Shared registry

One source of truth for library registrations

Configuration holder

App-wide settings loaded once

Logging facade

Single log channel avoids duplicate handlers

Connection pool

Heavy resources shared, not duplicated

❌ When Singleton Hurts You

Hidden global state

Any class can grab the singleton — coupling creeps in

Testing nightmare

Mocking singletons requires reflection hacks

Concurrency surprises

Mutable singletons need careful synchronization

SRP violation

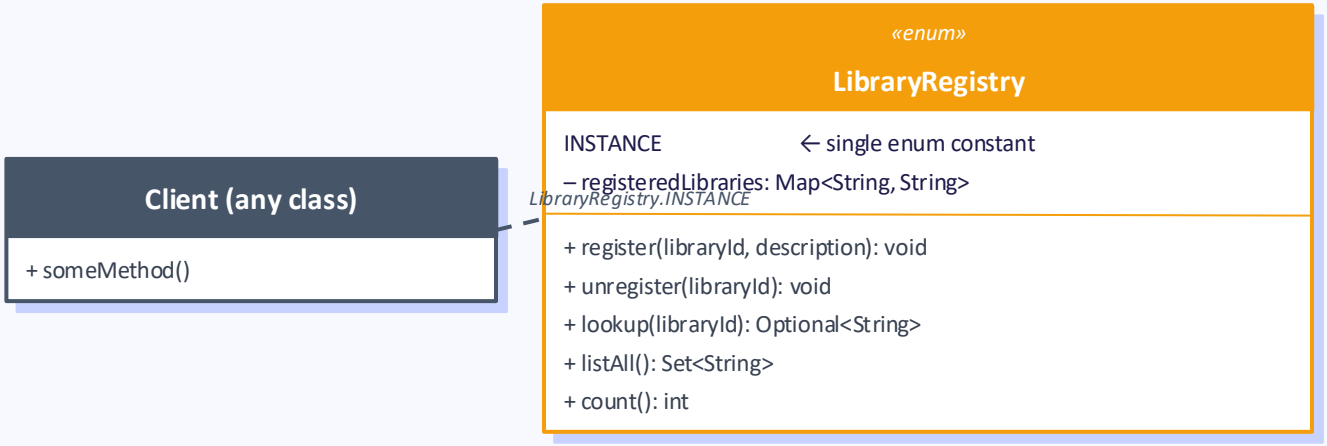
Singleton managing its own lifecycle + business logic

💡 SmartShelf uses LibraryRegistry.INSTANCE (enum) only for global library lookup — not for mutable state.

Class Diagram — Singleton Pattern

L6

Singleton



✅ Thread-safe
JVM class loading guarantees atomicity — no synchronised block needed.

✅ Serialisation-safe
Java serialisation preserves enum singletons automatically.

❌ Reflection-safe
Constructor.newInstance() throws IllegalArgumentException — blocked by JVM.

Factory Method Pattern

Lecture 6

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Creator (Abstract)

```
public abstract class
  MediaCreator {

  // Factory Method
  public abstract MediaItem
    createMedia(String title,
                String id);

  // Template method that
  // uses factory method
  public MediaItem
    orderAndSetup(String t,
                  String id) {
    MediaItem m =
createMedia(t,id);
    m.setAvailable(true);
    return m;
  }
}
```

Concrete Creator

```
public class BookCreator
  extends MediaCreator {

  private final String
defaultAuthor;

  public BookCreator(
    String defaultAuthor) {
    this.defaultAuthor =
      defaultAuthor;
  }

  @Override
  public MediaItem createMedia(
    String title, String isbn) {
    return new Book(title, isbn,
      defaultAuthor);
  }
}
```

Client Usage

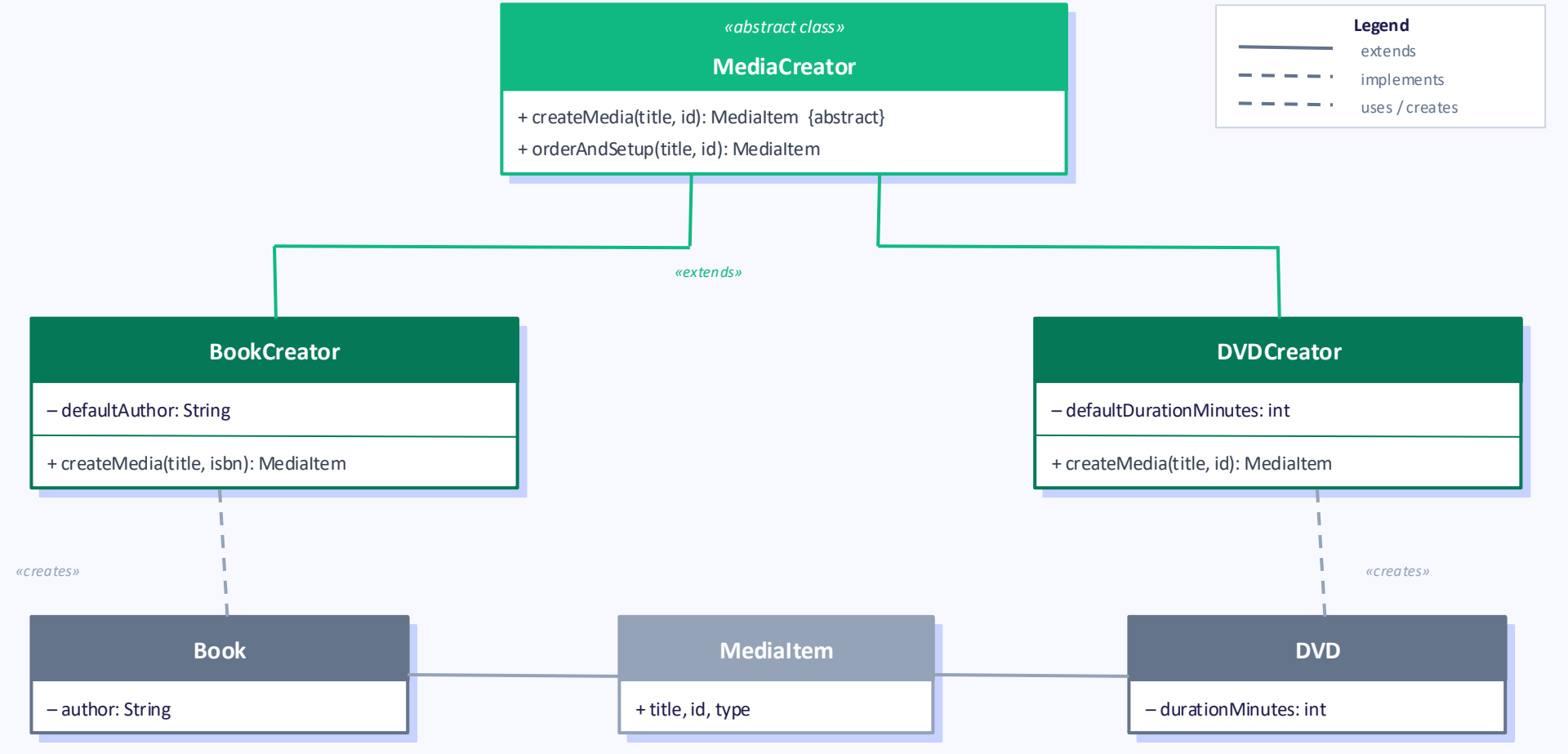
```
// Client works with Creator
MediaCreator creator =
  new BookCreator("Unknown");

MediaItem book =
  creator.orderAndSetup(
    "Clean Code", "978-0-13");

// Swap to DVDCreator later
// without changing client code!
MediaCreator dvdCreator =
  new DVDCreator(120);
MediaItem dvd =
  dvdCreator.orderAndSetup(
    "Inception", "DVD-001");
```

Key insight: Factory Method delays the decision of which class to create until a subclass or concrete creator is chosen. The client codes to the abstract type.

Class Diagram — Factory Method Pattern

L6

Factory Method



Abstract Factory Pattern

Lecture 6

Provide an interface for creating families of related objects without specifying their concrete classes.

MediaFactory (interface)

```
public interface MediaFactory {
  Book    createBook(String title,
                     String isbn,
                     String
author);
  DVD     createDVD(String title,
                    String id,
                    int
durationMin);
  Magazine createMagazine(
                    String title,
                    String issn,
                    int issueNo);
}
```

PhysicalMediaFactory

```
public class
  PhysicalMediaFactory
  implements MediaFactory {

  @Override public Book
  createBook(String t,
    String isbn, String a) {
    return new Book(t,isbn,a);
    // Physical: hardcover
  }
  @Override public DVD
  createDVD(String t,
    String id, int dur) {
    return new DVD(t,id,dur);
    // Physical: disc
  }
  // ...createMagazine()
}
```

DigitalMediaFactory

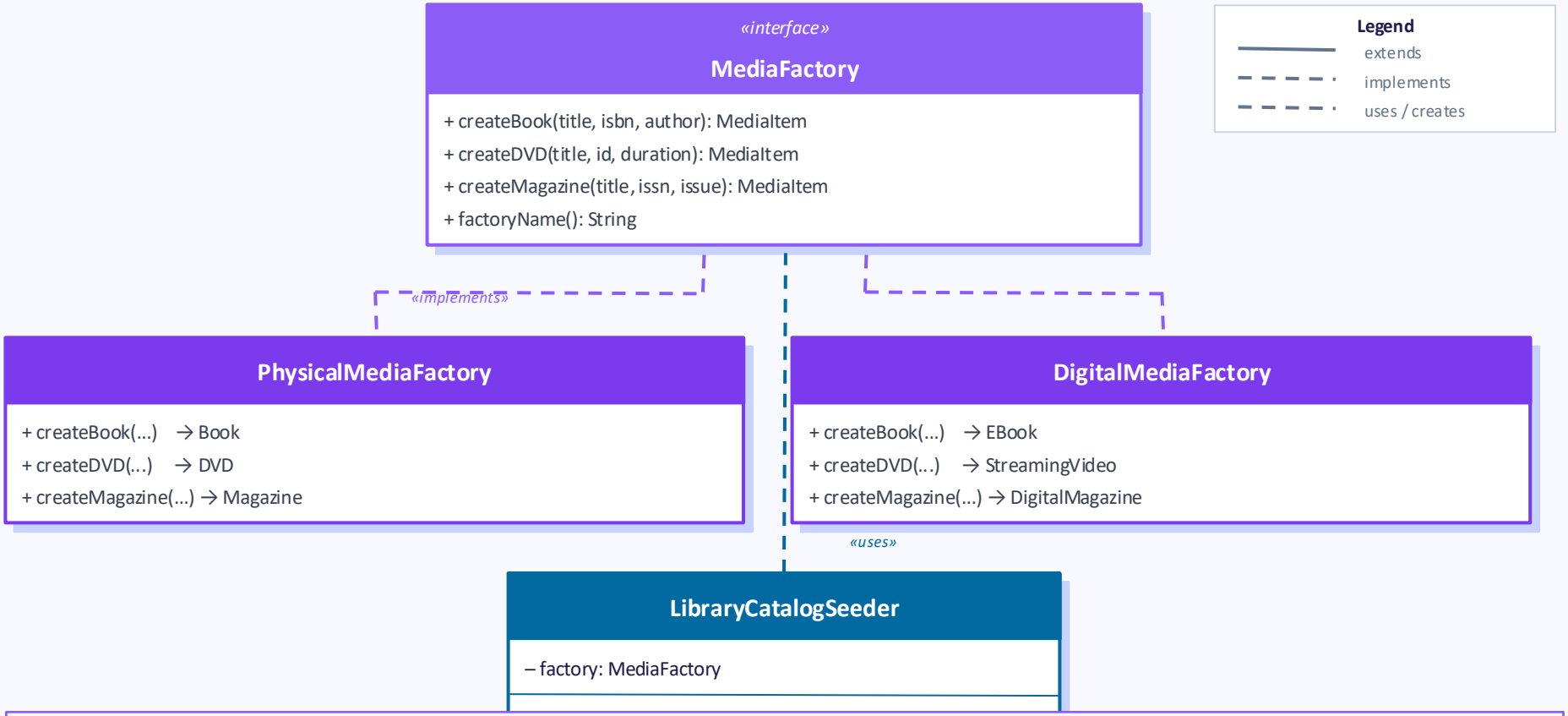
```
public class
  DigitalMediaFactory
  implements MediaFactory {

  @Override public Book
  createBook(String t,
    String isbn, String a) {
    return new EBook(t,isbn,a);
    // Digital: PDF/EPUB
  }
  @Override public DVD
  createDVD(String t,
    String id, int dur) {
    return new StreamingVideo(
      t, id, dur);
    // Digital: stream
  }
  // ...createMagazine()
```

Pattern	Factory Method	Abstract Factory
Focus	One product type	Families of related products
Structure	Subclassing	Composition (injects factory)

Class Diagram — Abstract Factory Pattern

L6

Abstract Factory



LibraryCatalogSeeder is IDENTICAL for both factories — swap the injected factory, get a completely different media family.

HOUR 3

Builder & Prototype Patterns

Constructing complex objects elegantly — and copying them cheaply

Builder Pattern — Complex Object Construction

Lecture 6

Separate the construction of a complex object from its representation, using a fluent step-by-step API.

❌ Problem — Telescoping Constructor

```
// Which arg is which??
new LoanRequest("M001", "ISBN-123",
  14, true, "Urgent request",
  LocalDate.now(), null, false);

// LoanRequest has 8 params — grows to 15?
// Optional params need null placeholders
// Impossible to read or maintain
```

✅ Solution — Builder Pattern

```
LoanRequest loan = new LoanRequest
  .Builder("M001", "ISBN-123")
  .durationDays(14)
  .priority(Priority.URGENT)
  .note("Student exam prep")
  .renewable(true)
  .build(); // validates + creates
```

Builder Structure in SmartShelf

Required Fields

memberId, isbn
Passed to Builder constructor
Always present — no defaults

Optional Fields

durationDays = 14
renewable = false
note = ""
priority = NORMAL

build() Validates

Checks: memberId not blank
isbn valid format
duration 1–180 days
Returns immutable LoanRequest

LoanRequestBuilder — Full Implementation


```
public final class LoanRequest {
  // Immutable fields
  private final String memberId;
  private final String isbn;
  private final int    durationDays;
  private final boolean renewable;
  private final String note;
  private final Priority priority;

  // Private constructor — only Builder creates it
  private LoanRequest(Builder b) {
    this.memberId    = b.memberId;
    this.isbn        = b.isbn;
    this.durationDays= b.durationDays;
    this.renewable   = b.renewable;
    this.note        = b.note;
    this.priority    = b.priority;
  }

  // Getters only — no setters (immutable!)
  public String getMemberId() { return memberId; }
  public String getIsbn()     { return isbn;     }
  // ...
}
```

```
// ── Static nested Builder ──────────
public static class Builder {
  // Required
  private final String memberId;
  private final String isbn;
  // Optional with defaults
  private int      durationDays = 14;
  private boolean  renewable    = false;
  private String   note         = "";
  private Priority priority     = Priority.NORMAL;

  public Builder(String memberId,
                 String isbn) {
    this.memberId = memberId;
    this.isbn     = isbn;
  }
  public Builder durationDays(int d) {
    this.durationDays = d; return this;
  }
  public Builder renewable(boolean r) {
    this.renewable = r; return this;
  }
  public Builder note(String n) {
    this.note = n; return this;
  }
  public Builder priority(Priority p) {
    this.priority = p; return this;
  }
  public LoanRequest build() {
    // Validate here before creating
    if (durationDays < 1 ||
        durationDays > 180)
      throw new ContractViolationException(
        "Duration must be 1-180 days");
    return new LoanRequest(this);
  }
}
```

Class Diagram — Builder Pattern

L6

Builder

LoanRequest.Builder (static nested)

– memberId: String ← required
– isbn: String ← required
– durationDays: int = 14
– renewable: boolean = false
– note: String = ""
– priority: Priority = NORMAL

+ Builder(memberId, isbn) ← validates required
+ durationDays(int): Builder
+ renewable(boolean): Builder
+ note(String): Builder
+ priority(Priority): Builder
+ build(): LoanRequest ← validates + creates

«creates»

LoanRequest (final)

– memberId: String
– isbn: String
– durationDays: int
– renewable: boolean
– note: String
– priority: Priority
– requestedDate: LocalDate

+ getMemberId(): String
+ getIsbn(): String
+ getDueDate(): LocalDate
– LoanRequest(Builder b) ← private!

🔒 LoanRequest has NO setters.
All fields final. Immutable after build().

«enum»

Priority

LOW · NORMAL · HIGH · URGENT

Prototype Pattern — Clone vs new

Lecture 6

Create new objects by copying an existing prototype. Avoids expensive initialisation by cloning.

When to use Prototype in SmartShelf:

📖 Book template library

A publisher provides default Book templates (title, genre, loan rules). New acquisitions start as a clone and customise the ISBN and copy number.

🏛 Branch seeding

Open a new library branch by cloning an existing branch's entire catalog scaffold (categories, policies, media types) instead of re-initialising everything.

🧪 Test data setup

Unit tests clone a pre-validated BookTemplate rather than calling the full constructor with 12 parameters each time.

Prototype Interface + Shallow vs Deep Copy

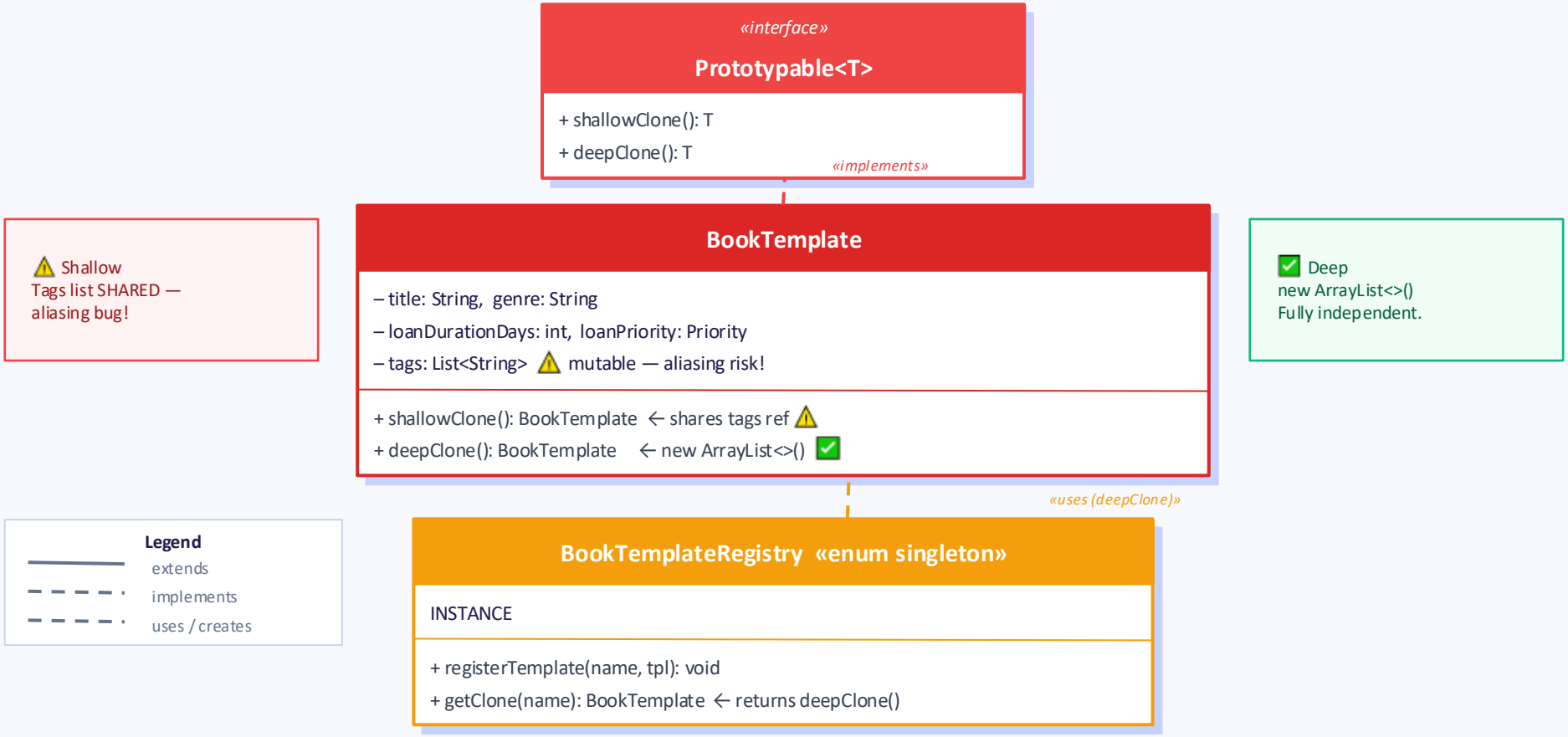
```
public interface Cloneable<T> {
  T clone(); // shallow copy
  T deepClone(); // deep — no shared refs
}
```

```
BookTemplate sci = new BookTemplate(
  "SciFi", 21, Priority.NORMAL);
BookTemplate copy = sci.deepClone();
copy.setTitle("Dune"); // original safe
```

Class Diagram — Prototype Pattern

L6

Prototype



HOUR 4

Workshop — Apply All 4 Patterns

Hands-on tasks: DIP refactoring, Singleton, Factory, Builder, Prototype

Workshop — Task A & Task B

Lecture 6

Task A: DIP Refactoring

30 min

1. Open DipDemo.java — observe Library directly instantiating EmailNotifier
2. Create interface INotifier with notify(Member m, String message)
3. Create EmailNotifier and SmsNotifier implementing INotifier
4. Refactor Library to accept INotifier via constructor injection
5. Write main() demonstrating swap: Email → SMS without changing Library

☆ Bonus: Create a CompositeNotifier that delegates to a List<INotifier>

Task B: Singleton LibraryRegistry

20 min

1. Implement LibraryRegistry as an enum singleton (INSTANCE pattern)
2. Add register(String id, Library lib) and lookup(String id) methods
3. Ensure no public constructor exists — only INSTANCE access
4. Demonstrate in main() that LibraryRegistry.INSTANCE == same object
5. Attempt to instantiate via reflection — catch the exception

☆ Bonus: Add an unregister() and list() method with stream + map

Workshop — Task C & Task D

Lecture 6

Task C: MediaFactory (Abstract Factory) — 30 min

1. Implement MediaFactory interface with createBook, createDVD, createMagazine
2. Build PhysicalMediaFactory — returns standard Book, DVD, Magazine
3. Build DigitalMediaFactory — returns EBook, StreamingVideo, DigitalMagazine
4. Write a LibraryCatalogSeeder that accepts any MediaFactory and seeds 3 items
5. Run seeder with both factories and print catalog to verify

☆ Bonus: Add a HybridMediaFactory that reads a property file to decide physical or digital

Task D: LoanRequestBuilder + BookPrototype — 30 min

1. Implement LoanRequest with inner Builder (required: memberId, isbn)
2. Add optional: durationDays, renewable, note, priority with defaults
3. build() must validate: duration 1–180, memberId not blank, isbn not blank
4. Implement BookTemplate with deepClone() — verify clones are independent
5. Create 3 different LoanRequests using Builder, print all fields

☆ Bonus: Implement a BookTemplateRegistry (Singleton) storing named BookTemplate prototypes

SmartShelf v0.6 — Project Structure

Lecture 6

Everything built in this lecture lives in the patterns/ and solid/ packages:

```
src/smartshelf/
├── patterns/
│   ├── singleton/
│   │   └── LibraryRegistry.java   ★ NEW
│   │       Enum singleton
│   ├── factory/
│   │   ├── MediaFactory.java      ★ NEW
│   │   │   Abstract factory iface
│   │   ├── PhysicalMediaFactory   ★ NEW
│   │   ├── DigitalMediaFactory    ★ NEW
│   │   ├── BookCreator.java       ★ NEW
│   │   │   Factory Method
│   │   └── MediaType.java         ★ NEW
│   ├── builder/
│   │   ├── Priority.java          ★ NEW
│   │   ├── LoanRequest.java       ★ NEW
│   │   └── LoanRequestBuilder     (inner)
│   └── prototype/
│       ├── Prototypable.java      ★ NEW
│       └── BookTemplate.java      ★ NEW
```

```
├── solid/
│   ├── INotifier.java         ★ NEW
│   │   DIP abstraction
│   ├── EmailNotifier.java     ★ NEW
│   ├── SmsNotifier.java       ★ NEW
│   └── DipDemo.java           ★ NEW
│       Before/After comparison
│
├── (all prior packages retained)
├── contract/
│   └── Contracts.java
├── exceptions/ (all 6 types)
├── generics/
│   ├── Catalog.java
│   ├── Pair.java
│   └── Result.java
├── Library.java     ← now DIP-clean
├── MediaItem.java
├── Book.java, DVD.java...
└── Main.java        ★ 8 new demos
```

COMING UP

Lecture 7: Structural Patterns

Adapter · Decorator · Facade · Composite

Adapter	Decorator	Facade	Composite
Wrap legacy ISBN validator — make old interfaces work with new code without changing either	Add logging, audit trail, and encryption to Library operations transparently	SmartShelfAPI — one clean entry point hiding the full subsystem complexity	Library network tree — branches and sub-branches treated uniformly

Lecture 6 Summary — Key Takeaways

Lecture 6

Dependency Inversion (DIP)

- High-level modules depend on abstractions, never on concrete classes
- Constructor injection is the cleanest implementation — no framework needed

Builder

- Ideal when an object has many optional parameters (> 4)
- Inner Builder + private constructor = immutable, validated object creation

Singleton — Use Sparingly

- Enum singleton is the safest: thread-safe, serialisation-safe, concise
- Avoid mutable singletons — they create hidden global state and test nightmares

Prototype

- Clone when construction is expensive or requires identical starting state
- deepClone() prevents aliasing — shared mutable references break isolation

Factory Patterns

- Factory Method: subclass decides which concrete type to create
- Abstract Factory: families of related objects without specifying concrete classes

Next: Lecture 7 — Structural Patterns
Adapter · Decorator · Facade · Composite

Questions & Discussion

SmartShelf v0.6 · SOLID Principles & Creational Patterns