LECTURE 3

Polymorphism, Dynamic Binding & Duck Typing

Advanced Object-Oriented Programming

Duration: 4 Hours | SmartShelf: Multiple Media Types

Today's Agenda

4 Hours — Making Objects Behave Differently at Runtime

Hour 1

What Is Polymorphism?

Types overview: compile-time (overloading) vs runtime (overriding)

Hour 2

Runtime Polymorphism & Dynamic Binding

Method overriding, late binding, MediaItem hierarchy in SmartShelf

Hour 3

Abstract Classes, Interfaces & Duck Typing

Abstract vs interface contracts, duck typing concept, design trade-offs

Hour 4

Workshop: SmartShelf Multi-Media Library

Tasks A–F: Journal, DVD, DigitalMedia, polymorphic search & display

Recap — SmartShelf v0.2

What we built in Weeks 1 & 2 + this week's new requirement

ISBN	Immutable value object — final, validated, safe to share
Book	Mutable with copy constructor — deep clones Author
Author	Mutable — copy constructor for deep cloning
Member	Encapsulated borrowedBooks, MemberType enum limits
Reservation	Deep copy snapshot of Book at reservation time

New Requirement: SmartShelf now lends Books, Journals, DVDs, and DigitalMedia. They share common behavior (borrow, return, display) but differ in specifics. How do we handle a mixed catalog?

Hour 1

What Is Polymorphism?

Polymorphism - Many Forms

One interface, multiple implementations — the heart of OOP flexibility

Polymorphism allows objects of different types to be treated through a common interface. The same method call produces different behavior depending on the actual object type at runtime.

Compile-Time (Static)

- Method Overloading
- Operator Overloading
- Resolved by compiler
- Same class, different signatures

```
search(String) vs search(ISBN)
```

Runtime (Dynamic)

- Method Overriding
- Dynamic Binding
- Resolved at runtime
- Subclass redefines parent method

```
mediaItem.display() → Book vs DVD
```

Compile-Time: Method Overloading

Same method name, different parameter lists — resolved at compile time

```
public class Library {

  // Search by title keyword
  public List<Book> search(String keyword) {
    return catalog.stream()
      .filter(b -> b.getTitle().contains(keyword))
      .collect(toList());
  }

  // Search by ISBN — different signature
  public Optional<Book> search(ISBN isbn) {
    return catalog.stream()
      .filter(b -> b.getIsbn().equals(isbn))
      .findFirst();
  }

  // Search by year range
  public List<Book> search(int fromYear, int toYear) {
    ...
  }
}
```

How It Works

Compiler picks the right method based on the parameter types in the call.

✓ Benefits

Readable API: one verb "search" for all query types. Type safety at compile time.

⚠ Limitation

Not "true" polymorphism — no runtime behavior change. Just syntactic convenience.

Runtime: Method Overriding

Subclass provides its own implementation of a parent method — same signature

```
abstract class MediaItem {
  public String display() {
    return isbn + ": " + title;
  }
}

class Book extends MediaItem {
  @Override
  public String display() {
    return title + " by " + author
           + " (" + pageCount + "pp)";
  }
}

class DVD extends MediaItem {
  @Override
  public String display() {
    return title + " [" + duration + "min]";
  }
}
```

Override Rules

✓ Same method name

✓ Same parameters

✓ Same or covariant return

✓ Access ≥ parent

✓ @Override annotation

Key Difference

The JVM decides at runtime which display() to call based on the actual object type, not the declared type.

Hour 2

Runtime Polymorphism & Dynamic Binding

Dynamic Binding (Late Binding)

The JVM resolves the method call at runtime, not compile time

```
// Declared type: MediaItem
// Actual type: Book
MediaItem item = new Book(...);

item.display();
// → calls Book.display(), NOT MediaItem.display()
// JVM looks at ACTUAL type at runtime
```

How JVM Resolves

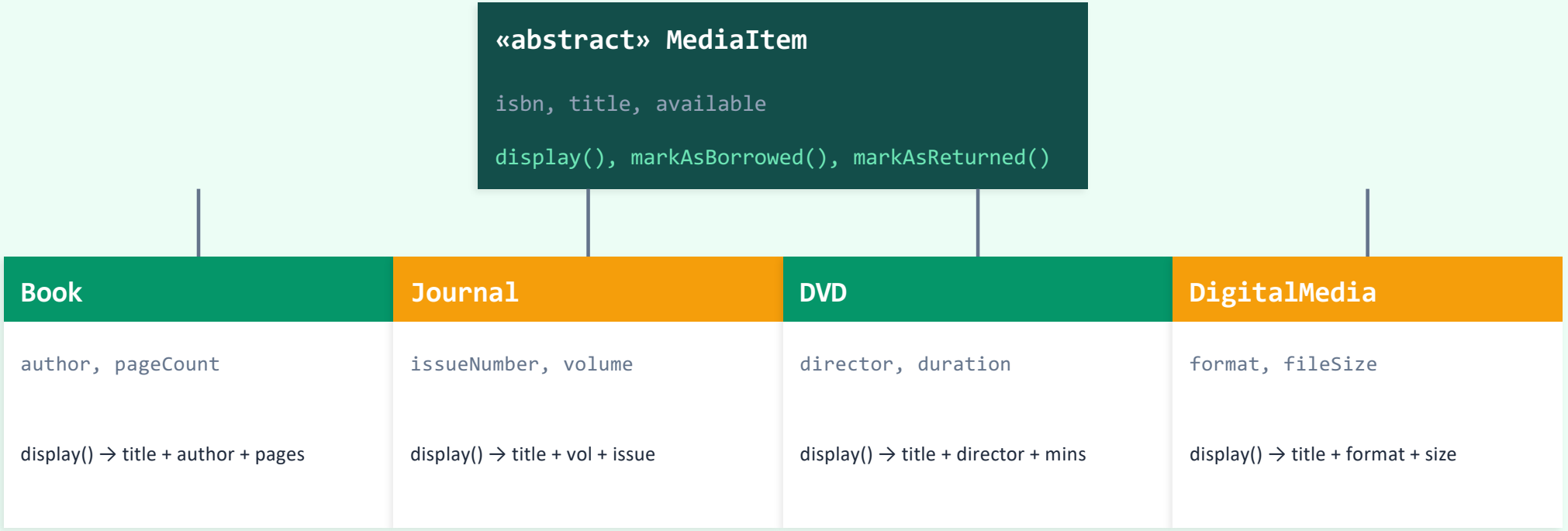
1. Compiler: checks MediaItem has display()
2. Runtime: JVM sees actual type is Book
3. JVM: dispatches to Book.display()

```
// Polymorphic collection — ONE loop handles ALL types
List<MediaItem> catalog = List.of(book, journal, dvd, ebook);

for (MediaItem item : catalog) {
  System.out.println(item.display());
// Each type's own display()
}
```

SmartShelf: MediaItem Hierarchy

Abstract parent class with four concrete subtypes



Each subtype overrides display() — the caller doesn't need to know which type it is.

Polymorphic Collections in Action

One loop, four different behaviors — the power of runtime polymorphism

```
List<MediaItem> catalog = new ArrayList<>();
catalog.add(new Book(isbn1, "Effective Java", ...));
catalog.add(new Journal(isbn2, "IEEE SW", ...));
catalog.add(new DVD(isbn3, "Clean Code", ...));
catalog.add(new DigitalMedia(isbn4, "SICP", ...));

// One loop — JVM dispatches to correct display()
for (MediaItem item : catalog) {
  System.out.println(item.display());
}

// Output:
// Effective Java by J.Bloch (416pp)
// IEEE SW Vol.39 Issue.2
// Clean Code [90min] dir: R.Martin
// SICP [PDF, 4.2MB]
```

✓ Open/Closed Principle

Add new media types (AudioBook, Thesis) without changing the loop, Library class, or display logic. Just extend MediaItem.

Before Polymorphism

```
if (item instanceof Book)
  displayBook(item);
else if (item instanceof DVD)
  displayDVD(item);
// + every new type = new branch
```

LO1: Evaluate — polymorphism eliminates conditional type checks

The instanceof Anti-Pattern

Why type-checking conditionals defeat the purpose of polymorphism

❌ BAD — instanceof chain

```
void displayItem(MediaItem m) {
  if (m instanceof Book) {
    Book b = (Book) m;
    print(b.getAuthor());
  } else if (m instanceof DVD) {
    DVD d = (DVD) m;
    print(d.getDirector());
  } // + every new type...
}
```

Violates Open/Closed: every new type = code change

✓ GOOD — polymorphic dispatch

```
void displayItem(MediaItem m) {
  System.out.println(
    m.display()
  );
  // JVM calls the right version
  // No casting, no conditionals
  // New types: zero changes here
}
```

Follows Open/Closed: new type = new class only

Critical Evaluation — LO1

Polymorphism has utility AND pitfalls

Aspect	Utility	Pitfall
Overloading	Clean API — one verb, many params	Not true polymorphism, just convenience
Overriding	Runtime flexibility, Open/Closed	Deep hierarchies become fragile
Dynamic Binding	Add types without changing callers	Debugging harder — which impl runs?
Poly Collections	One loop for all types	Type info lost — need override discipline

LO1: Critically evaluate OOP principles

In assessments: always argue both sides. Polymorphism reduces conditionals but adds hierarchy complexity. Show awareness of trade-offs with concrete SmartShelf examples.

Hour 3

Abstract Classes, Interfaces & Duck Typing

Abstract Classes

Partial implementation — shared state + enforced method contracts

```
public abstract class MediaItem {
  private final ISBN isbn;      // shared state
  private String title;
  private boolean available;

  // Concrete method – shared behavior
  public void markAsBorrowed() {
    if (!available) throw ...;
    available = false;
  }

  // Abstract – each subtype MUST implement
  public abstract String display();
}
```

Key Properties

- Cannot be instantiated
- Can have fields + constructors
- Mix of concrete & abstract methods
- Subclass must implement abstract methods

✓ Use When

Subtypes share state AND behavior. MediaItem holds isbn, title, available for ALL subtypes.

⚠️ **Limitation:** Single inheritance only — a class can extend only one abstract class.

Interfaces — Pure Contracts

Define what objects can do, not how — supports multiple contracts

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

interface Reservable {
  Reservation reserve(String memberId);
}

interface Notifiable {
  void notifyMember(String message);
}

// A class can implement MULTIPLE interfaces
class Library implements Searchable,
                         Reservable { ... }
```

Key Properties

- No instance fields
- All methods abstract (by default)
- Multiple inheritance of type
- Since Java 8: default methods

✓ Use When

Define capabilities that cut across hierarchies. Library is Searchable AND Reservable.

🗣️ Discussion

Should MediaItem be an abstract class or an interface? What are the trade-offs?

Abstract Class vs Interface — Comparison

When to use which in SmartShelf

Feature	Abstract Class	Interface
Fields	✓ Instance fields	✗ Constants only
Constructors	✓ Yes	✗ No
Methods	Mix: concrete + abstract	Abstract (+ default since J8)
Inheritance	Single only	Multiple
Access Modifiers	Any	public (implicit)
SmartShelf Use	MediaItem (shared state)	Searchable, Reservable
Design Intent	"is-a" relationship	"can-do" capability

💡 **Rule of Thumb:** Use abstract class when subtypes share state (fields). Use interfaces for capabilities that cut across unrelated classes. Often you'll use both together.

Duck Typing — Concept & Contrast

"If it walks like a duck and quacks like a duck, it is a duck"

Java — Nominal Typing

```
// Must explicitly declare type
class Library implements Searchable {
  public List<MediaItem> search(
      String q) { ... }
}

// Type checked at compile time
Searchable s = new Library();
```

✓ Compile-time safety ⚠ Must declare implements

"You must say you're a duck"

Python — Duck Typing

```
# No interface declaration needed
class Library:
  def search(self, query):
    ...

# Just call the method
# If it has search(), it works
def find(searchable, q):
  return searchable.search(q)
```

✓ Flexible, less boilerplate ⚠ Runtime errors

"If it quacks, that's enough"

💡 **Java trend:** Java is moving toward duck-typing-like behavior with var, sealed interfaces, and pattern matching (Java 17+).

SmartShelf: Combining Abstract + Interface

Real designs use both — abstract for shared state, interfaces for capabilities

```
// Abstract class – shared state + common behavior
abstract class MediaItem {
  private final ISBN isbn;
  private String title;
  private boolean available;

  public void markAsBorrowed() { ... }   // concrete – shared
  public abstract String display();       // abstract – each overrides
}

// Interfaces – cross-cutting capabilities
interface Downloadable { String getDownloadUrl(); }
interface PhysicalItem  { String getShelfLocation(); }

// Concrete classes combine abstract + interfaces
class Book extends MediaItem implements PhysicalItem { ... }
class DigitalMedia extends MediaItem implements Downloadable { ... }
```

Hour 4

Workshop: SmartShelf Multi-Media Library

Workshop Tasks A – C

Build the hierarchy — 45 minutes

Task A

Refactor Book → abstract MediaItem parent

Extract isbn, title, available into MediaItem. Make display() abstract. Book extends MediaItem, keeps author and pageCount.

Task B

Create Journal and DVD subclasses

Journal: issueNumber, volume. DVD: director, durationMinutes. Each overrides display() with type-specific formatting.

Task C

Create DigitalMedia subclass + Downloadable

DigitalMedia: format (PDF/EPUB), fileSize. Implements Downloadable interface. getDownloadUrl() returns simulated link.

After Task C: List<MediaItem> can hold Book, Journal, DVD, DigitalMedia — test with a mixed catalog.

Workshop Tasks D – F

Wire polymorphic behavior — 45 minutes

Task D

Update Library to use List<MediaItem>

Refactor catalog from List<Book> → List<MediaItem>. addItem(), findByIsbn(), borrowItem() all work polymorphically.

Task E

Implement polymorphic search & display

Library.displayAll() loops catalog, calls display() on each — no instanceof! Add overloaded search(String), search(ISBN).

Task F

Full Integration Demo

Create 2 Books, 1 Journal, 1 DVD, 1 DigitalMedia. Borrow, search, display all. Prove: adding a 6th type requires zero changes to Library.

✓ No instanceof ✓ No type casts ✓ display() dispatches via JVM ✓ Open/Closed

Running the Workshop Code

SmartShelf v0.3 project structure

```
SmartShelf/
├── run.sh
├── src/smartshelf/
│   ├── ISBN.java            Week 1
│   ├── Author.java          Week 2
│   ├── MediaItem.java        Task A ★
│   ├── Book.java             Task A ★
│   ├── Journal.java          Task B ★
│   ├── DVD.java              Task B ★
│   ├── DigitalMedia.java     Task C ★
│   ├── Downloadable.java     Task C ★
│   ├── Library.java          Task D+E ★
│   └── Main.java             Task F ★
└── bin/
```

What's New This Week

★ MediaItem abstract class
★ 3 new subclasses
★ Downloadable interface
★ Library uses List<MediaItem>
★ Polymorphic display & search

What You'll See

Each display() output formatted differently. One loop handles all types. Search returns mixed results.

What's Coming Next

Week 4: Generics & Liskov Substitution Principle

SmartShelf: Type-Safe Collections

SmartShelf will use generic classes — Catalog<T extends MediaItem>, Pair<K,V> — to ensure type safety at compile time while maintaining polymorphic behavior. We'll explore the Liskov Substitution Principle: when can a subtype safely replace its parent?

📚 Preparation

Read Head First Java, Ch. 16 (Generics). Complete Tasks A–F and push to repo. Review Java Generics tutorial on Oracle docs.

Lecture 3 Summary

🔄 Overloading: compile-time — same name, different params

🎭 Overriding: runtime — subclass redefines parent method

⚡ Dynamic binding: JVM dispatches to actual type's method

📋 Abstract classes: shared state + enforced contracts

🔌 Interfaces: cross-cutting capabilities, multiple inheritance

🦆 Duck typing: behavior over declarations (Python vs Java)

🏗️ SmartShelf v0.3: MediaItem hierarchy + polymorphic catalog

Questions?

Next Week: Generics & Liskov Substitution Principle