LECTURE 2

# Cloning: Deep & Shallow Copying

Advanced Object-Oriented Programming

Duration: 4 Hours | Building on SmartShelf v0.1

# Today's Agenda

4 Hours — Understanding How Objects Are Copied

**Hour 1**

## References, Aliasing & Object Identity

Why copying objects is harder than copying primitives

**Hour 2**

## Shallow Cloning — Cloneable & clone()

Java's built-in cloning mechanism and its pitfalls

**Hour 3**

## Deep Cloning — Copy Constructors & Beyond

Safe copying of object graphs with nested mutable state

**Hour 4**

## Workshop: SmartShelf Reservation System

Build a reservation feature requiring proper deep copies

# Recap — SmartShelf v0.1 (Last Week)

| | |
|---|---|
| **ISBN** | Immutable value object — final class, no setters, validated |
| **Book** | Mutable — encapsulated fields, controlled state transitions |
| **Member** | Encapsulated borrowedBooks list, defensive unmodifiable view |
| **Library** | Catalog & member management, borrowBook() validation chain |

### New Requirement This Week

Students can reserve books that are currently borrowed. The reservation needs a snapshot of the book's info — but what happens if the original book's state changes?

# Hour 1

References, Aliasing & Object Identity

# Primitives vs Object References

The fundamental difference that makes cloning necessary

## Primitives — Copy by Value

```
int a = 42;
int b = a;     // Copies the value

b = 100;

// a is still 42
// b is 100
// Completely independent!
```

Each variable holds its own copy of the data

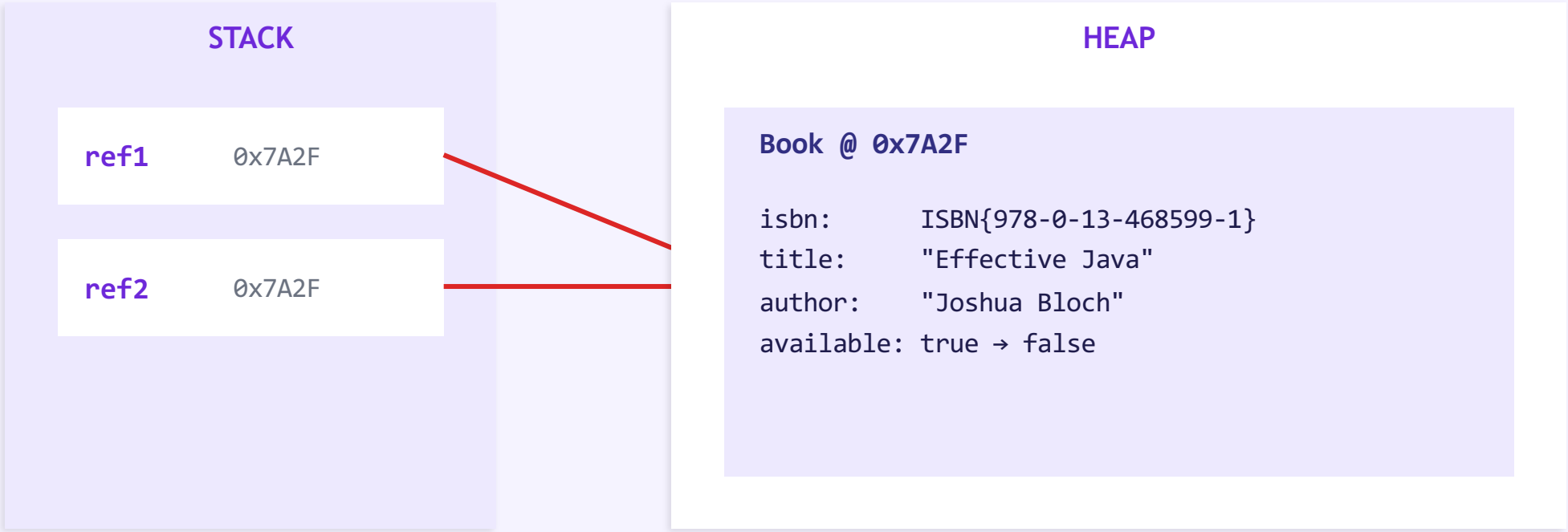
## Objects — Copy by Reference

```
Book a = new Book(isbn, ...);
Book b = a;   // Copies the REFERENCE

b.markAsBorrowed();

// a.isAvailable()? false!
// Both point to SAME object
// Changes via b affect a!
```

Both variables point to the same object in heap memory

# Memory Layout: Aliasing in Action

SmartShelf: Two references to the same Book object




**ref1 and ref2 hold the same memory address → changing state via either reference affects both**

# Identity (==) vs Equality (.equals())

| Operator | What It Checks | SmartShelf Example | Result |
|---|---|---|---|
| `==` | Same object in memory (same address) | `ref1 == ref2  (both point to same Book)` | **true** |
| `==` | Same object in memory | `isbn1 == isbn2  (different objects, same value)` | **false** |
| `.equals()` | Same value / logical equality | `isbn1.equals(isbn2)  (same ISBN value)` | **true** |

**SmartShelf: Why this matters for cloning**

```
Book original = library.findBookByIsbn(isbn).get();
Book copy = original;                              // NOT a clone — just another reference
copy.markAsBorrowed();                             // original is also marked as borrowed!
// We need a REAL copy — a new object with the same data
```

# The Copying Spectrum

Three levels of "copying" an object

## Reference Copy

DANGEROUS

`= assignment`

No new object created. Both variables point to the same object. Changes via one are visible through the other.

## Shallow Clone

PARTIAL

`clone()`

New object created. Primitive fields are copied. Object fields still share references to the same nested objects.

## Deep Clone

SAFE

`Copy constructor`

New object created. All nested objects are also recursively copied. Completely independent from original.

# Hour 2

Shallow Cloning — Cloneable & clone()

# Java's Cloneable Interface

The built-in (but controversial) cloning mechanism

**SmartShelf: Book implements Cloneable**

```
public class Book implements Cloneable {
  private final ISBN isbn;      // immutable
  private String title;         // mutable
  private Author author;        // mutable object!
  private boolean available;

  @Override
  public Book clone() {
    try {
      return (Book) super.clone();
    } catch (CloneNotSupportedException e){
      throw new AssertionError();
    }
  }
}
```

## How It Works

- Cloneable is a marker interface (no methods)
- super.clone() creates a bitwise copy
- Primitive fields are copied by value
- Object fields are copied by reference

⚠ **Result: new Book, but same Author object shared between original and clone**

# Shallow Clone — Memory Diagram

`Book clone = original.clone(); — What actually happens in memory`

**original: Book @ 0x7A2F**

```
isbn:      → ISBN @ 0xBB10 ✓
title:     "Effective Java" ✓
author:    → Author @ 0xCC20 ⚠️
available: true ✓
```

**clone: Book @ 0xDD30 (NEW)**

```
isbn:      → ISBN @ 0xBB10 ✓
title:     "Effective Java" ✓
author:    → Author @ 0xCC20 ⚠️
available: true ✓
```

**ISBN @ 0xBB10**

*SAFE — ISBN is immutable*

**Author @ 0xCC20 {name: "Joshua Bloch"}**

*SHARED — changing name via clone also changes original!*

# ⚠️ Shallow Clone Pitfall — Live Demo

The bug that shallow cloning creates in SmartShelf

```
// Create a book with a mutable Author
Author author = new Author("Joshua Bloch", "joshua@example.com");
Book original = new Book(isbn, "Effective Java", author, 2018, "Programming");

// Shallow clone
Book clone = original.clone();

// Verify: different objects
System.out.println(original == clone);                    // false ✓

// THE BUG: mutate author via the clone
clone.getAuthor().setEmail("hacked@evil.com");

// Check original — it's been affected!
System.out.println(original.getAuthor().getEmail());     // "hacked@evil.com" ✗

// Both Book objects share the SAME Author reference
System.out.println(original.getAuthor() == clone.getAuthor());  // true — PROBLEM!
```

# When Is Shallow Clone Sufficient?

| Field Type | Shallow Clone Safe? | SmartShelf Example | Why |
|---|---|---|---|
| **Primitive (int, boolean)** | ✓ Yes | `available, publishYear` | Copied by value |
| **Immutable object** | ✓ Yes | `ISBN, String` | Cannot be modified |
| **Mutable object** | ✗ No | `Author, List<Book>` | Shared reference = aliasing bug |

## Rule of Thumb

Shallow clone is safe ONLY when all object-typed fields are either immutable or you intentionally want shared state. If any mutable object field exists, you need deep cloning — or you'll create aliasing bugs that are extremely hard to debug.

# Hour 3

Deep Cloning — Copy Constructors & Beyond

# Copy Constructor Pattern

The recommended alternative to clone() — explicit, safe, and flexible

### SmartShelf: Author with Copy Constructor

```
public class Author {
  private String name;
  private String email;

  // Normal constructor
  public Author(String name, String email) {
    this.name = name;
    this.email = email;
  }

  // COPY CONSTRUCTOR — creates deep copy
  public Author(Author other) {
    this.name  = other.name;  // String is immutable
    this.email = other.email; // String is immutable
  }
}
```

### Advantages Over clone()

- No Cloneable needed
- No CloneNotSupportedException
- No unsafe cast from Object
- You control exactly what gets copied
- Works with final fields
- Easy to read and debug
- Can convert between types

# Deep Clone: Book Copy Constructor

Recursively copying nested mutable objects

```
public class Book {
    private final ISBN isbn;         // Immutable → safe to share
    private String title;            // String immutable → safe to share
    private Author author;           // MUTABLE → must deep copy!
    private boolean available;

    // DEEP COPY CONSTRUCTOR
    public Book(Book other) {
        this.isbn      = other.isbn;               // Safe — ISBN is immutable
        this.title     = other.title;              // Safe — String is immutable
        this.author    = new Author(other.author); // DEEP COPY — new Author object
        this.available = other.available;           // Safe — primitive
    }

    // Usage:
    // Book deepCopy = new Book(original);
    // deepCopy.getAuthor() != original.getAuthor()  → true (independent!)
}
```

# Deep Clone — Memory Diagram

`Book deepCopy = new Book(original);` — Completely independent

**original: Book @ 0x7A2F**

```
isbn:   → ISBN @ 0xBB10 (shared — immutable ✓)
author: → Author @ 0xCC20
```

**deepCopy: Book @ 0xEE40 (NEW)**

```
isbn:   → ISBN @ 0xBB10 (shared — immutable ✓)
author: → Author @ 0xFF50 (NEW!)
```

**Author @ 0xCC20** `{"Joshua Bloch", "josh@real.com"}`

*Original's own Author — independent*

**Author @ 0xFF50** `{"Joshua Bloch", "josh@real.com"}`

*Deep copy's own Author — independent*

**ISBN @ 0xBB10 — safely shared (immutable, no need to copy)**

# Deep Cloning Collections

SmartShelf: Copying a Member's borrowed books list

```
// ✗ WRONG — shallow list copy
List<Book> copy = new ArrayList<>(original);
// New list, but same Book references inside!

// ✓ CORRECT — deep copy each element
List<Book> deepCopy = original.stream()
    .map(book -> new Book(book))  // copy ctor
    .collect(Collectors.toList());

// ✓ ALTERNATIVE — for loop
List<Book> deepCopy2 = new ArrayList<>();
for (Book b : original) {
    deepCopy2.add(new Book(b)); // deep copy
}
```

## Key Insight

new ArrayList<>(list) only copies the references, not the objects

**To deep copy a collection:**

- Create a new list
- Deep copy each element
- Add copies to new list

*If elements are immutable (like ISBN), a shallow list copy is sufficient.*

# Comparing Cloning Approaches

| Approach | Deep? | Final Fields? | Type Safe? | Performance | Verdict |
|---|---|---|---|---|---|
| `= assignment` | ✗ No | N/A | N/A | Instant | **Not a copy** |
| `clone()` | Shallow only | ✗ No | ✗ Needs cast | Fast | **Avoid** |
| `Copy constructor` | ✓ Deep | ✓ Yes | ✓ Yes | Fast | **Recommended** |
| `Serialization` | ✓ Deep | ✓ Yes | ✓ Yes | Slow | **Last resort** |

### From Effective Java (Item 13)

Joshua Bloch recommends avoiding the Cloneable interface entirely. Instead, use copy constructors or static factory methods — they're clearer, more flexible, and don't require implementing a broken interface.

# SmartShelf: What Needs Deep Cloning?

| Class | Cloning Strategy | Reasoning |
|---|---|---|
| **ISBN** | **No clone needed** | Already immutable — safe to share references |
| **Author** | **Copy constructor** | Mutable (email can change) — must deep copy |
| **Book** | **Copy constructor** | Contains mutable Author — needs deep copy for reservations |
| **Member** | **Copy constructor** | Contains mutable List<Book> — deep copy needed for snapshots |
| **Reservation** | **Uses deep copied Book** | NEW CLASS — stores a snapshot of the book at reservation time |

## Design Principle

Immutable objects are the best friends of cloning. The more immutable your design, the less deep copying you need. This is why we made ISBN immutable in Week 1 — it pays off now.

# Hour 4

Workshop: SmartShelf Reservation System

# Workshop Exercise 1: Author & Cloning Infrastructure

Build the cloning foundation — 40 minutes

**Task A**

## Create the Author class (mutable)

Fields: name, email. Setters with validation. Implement a copy constructor: Author(Author other)

**Task B**

## Add copy constructors to Book

Implement Book(Book other) — share ISBN (immutable), deep copy Author. Keep Cloneable for comparison.

**Task C**

## Demonstrate the shallow clone bug

Clone a Book, modify the Author via the clone, verify original is affected. Then show deep copy fixes it.

**Key Question: After deep copy, are original.getAuthor() == copy.getAuthor()? What about original.getIsbn() == copy.getIsbn()?**

# Workshop Exercise 2: Reservation System

Apply cloning to a real feature — 50 minutes

**Task D**

## Create the Reservation class

Fields: reservationId, memberId, bookSnapshot (deep copy of Book), reservationDate, status. The bookSnapshot must be independent.

**Task E**

## Add reserveBook() to Library

Validate: member exists, book exists, book is currently borrowed. Create a Reservation with a deep copy of the book's current state.

**Task F**

## Full Integration Demo

Borrow books, make reservations, modify the original book's author — prove the reservation snapshot is unaffected.

**This is the real-world motivation for deep cloning: preserving a historical snapshot even when the original changes.**

# What's Coming Next

## Week 3: Polymorphism, Dynamic Binding & Duck Typing

### SmartShelf Evolution: Multiple Media Types

SmartShelf will handle Books, Journals, DVDs, and DigitalMedia — all extending MediaItem.

We'll use runtime polymorphism to process mixed collections and dynamic binding for search.

How do you search across different media types with different fields? Polymorphism solves this.

Preparation: Read Head First Java, Ch. 7–8 (Inheritance & Interfaces). Complete Tasks A–F from today and push to your repository.

# Lecture 2 Summary

- Reference copy (=) creates aliases, not copies
- Shallow clone: new object, shared nested references
- Deep clone via copy constructors: safe & recommended
- Immutable objects (ISBN) never need cloning
- Reservation system built with deep copy snapshots
- Design for immutability → less cloning needed

*SmartShelf v0.2 — Now with Author, deep cloning, and reservations*

# Questions?

Next Week: Polymorphism, Dynamic Binding & Duck Typing