برنامه نویسی پیشرفته

رفع اشکال: جلسه ۲

Object composition (has-a relationships).
String manipulation, immutability, common methods and enums.
Writing and generating Javadoc comments for code documentation, built-in packages.

Object Composition – Introduction

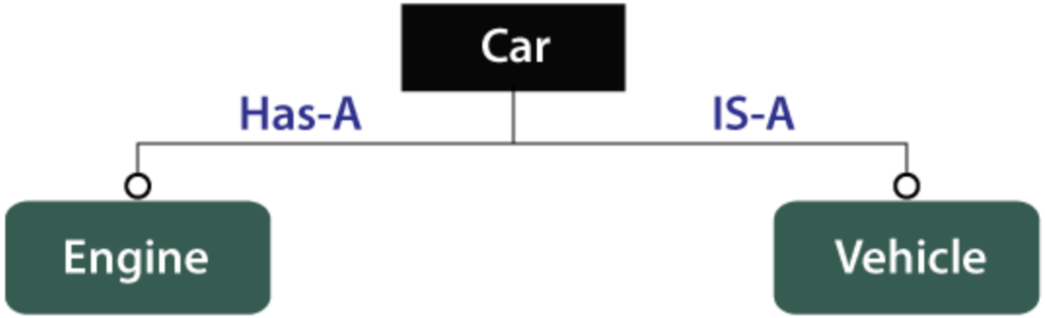
Definition: A *has-a* relationship means one class contains a reference to another class (field).

A University "has" multiple Colleges.

Why :

- Enhances **modularity**
- Encourages **code reuse**
- Simplifies **testing** (each class can be tested independently).

Don't confuse *has-a* with *is-a* (inheritance)



KEY CHARACTERISTICS

Strong Coupling: Sometimes one object's data closely depends on the other.

```
class Book {
    private String title;
    private String author;
    // ... }
class Shelf {
    private List<Book> books = new ArrayList<>();
    // ...}
class Library {
    private List<Shelf> shelves = new ArrayList<>();
    // ...}
```

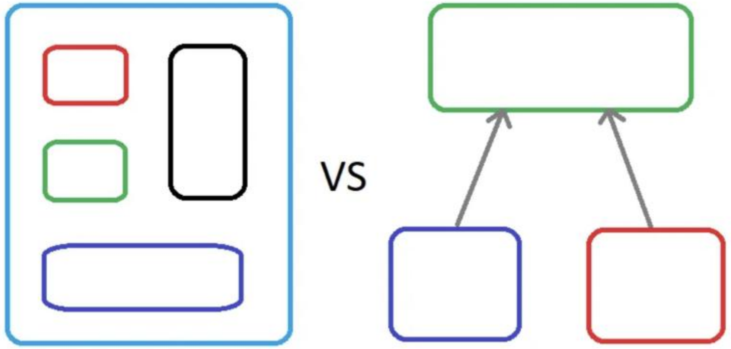
Each Library can contain multiple Shelfs, and each Shelf can contain multiple Books.

NOTE

Common Mistake : Composition can be overkill if the objects truly don't belong to one another.
For instance, a House having a Cat might be debatable if the cat is more of a "resident" than a tightly composed entity.

each level of composition has its own business logic :

```
class Shelf {
    private List<Book> books;
    private final int capacity; // Business logic: max books allowed per shelf
    public Shelf(int capacity) {
        this.capacity = capacity;
        this.books = new ArrayList<>();
    }
    // Method to add a book while respecting the shelf capacity
    public boolean addBook(Book book) {
        if (books.size() < capacity) {
            books.add(book);
            return true;
        } else {
            System.out.println("Shelf is full! Cannot add more books.");
            return false;
        }
    }
}
```

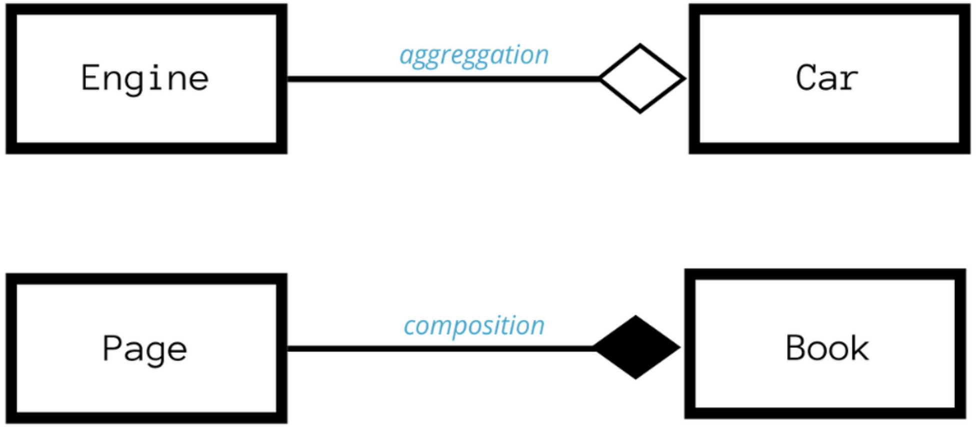


AGGREGATION

Aggregation is a weaker form of composition where one object contains another, but the contained object can exist independently.
It represents a "has-a" relationship, just like composition, but with **less strict ownership**.
If the container object is **destroyed**, the contained object **can still exist.**

```
class Player {
    private String name;

    public Player(String name) {
        this.name = name;}
    public String getName() { return name; }}
class Team {
    private List<Player> players = new ArrayList<>(); //
Players can exist outside Team
    public void addPlayer(Player player) {
        players.add(player);
    }
    public void showPlayers() {
        for (Player p : players) {
            System.out.println("Player: " + p.getName());}
    }
}
```



OBJECT INTERACTION

Definition:When an object (client) calls methods on another object (server).

```
class NumberDisplay {
    private int value;
    private int limit;
    public NumberDisplay(int limit) {
        this.limit = limit;
        this.value = 0;
    }
    public void increment() {
        value = (value + 1) % limit;
    }
    public int getValue() {
        return value;}
}
```

```
class ClockDisplay {
    private NumberDisplay hours;
    private NumberDisplay minutes;

    public ClockDisplay() {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
    }

    public void timeTick() {
        minutes.increment();
        if (minutes.getValue() == 0) {
            hours.increment();
        }
    }
}
```

ClockDisplay **has-a** relationship with NumberDisplay for **minutes** and **hours**.timeTick() updates **minutes**, and if minutes reset (00), hours increment.

QUESTIONS

- A) What situations in the library example create null references and how can they be prevented?
- B) ClockDisplay & NumberDisplay / Library Contains Books , Aggregation Or Composition ?

ANSWERS

A) When the Library is created, it starts empty. We must handle this properly.

- Calling listAllBooks() on an empty library : Might lead to NullPointerException if shelves was null.
  Fix: Always initialize shelves in the constructor as an empty list (new ArrayList<>()).

  ```
  this.shelves = new ArrayList<>();
  ```

- Iterating through an empty list : Printing books might produce an empty loop.
  Fix: Explicitly check shelves.isEmpty() and notify the user.

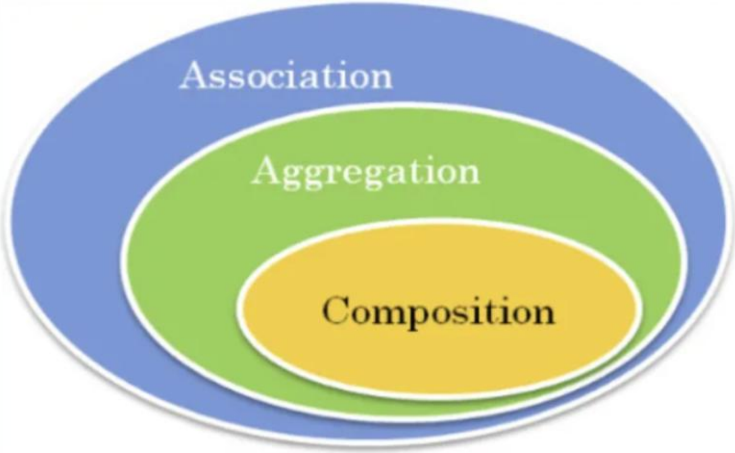
- Accessing getBooks() when no books exist
  Fix: Use .size() checks before accessing elements.

B) ClockDisplay depends on NumberDisplay

If the parent dies, the contained object has nowhere to belong

The parent is responsible for creating the object in composition.

Library contains Books, but the books exist separately too



STRINGS IN JAVA – OVERVIEW

Strings Are Objects , belonging to class String

Immutable: Once created, their state cannot be modified.
Typical Creation :

1) Literal : `String s = "Hello” ;`

2) Constructor : `String s = new String("Hello") ;`

Pitfall : Using Constructor method repeatedly can hurt performance because each call creates a new String on the heap rather than reusing the string pool.

STRING IMMUTABILITY – WHY IT MATTERS

- **Memory & the String Pool**: Java stores literal strings in a special pool to avoid duplicates.
- **Concurrency**: Safe in multi-threaded contexts since it can’t be changed once created.
- **“Modifying” Strings**: Actually returns a new object (e.g., s.concat("!")).

```
String str = "Hello";
str.concat(" World"); // Creates a new String, but doesn't modify str
System.out.println(str); // Still prints "Hello"
```

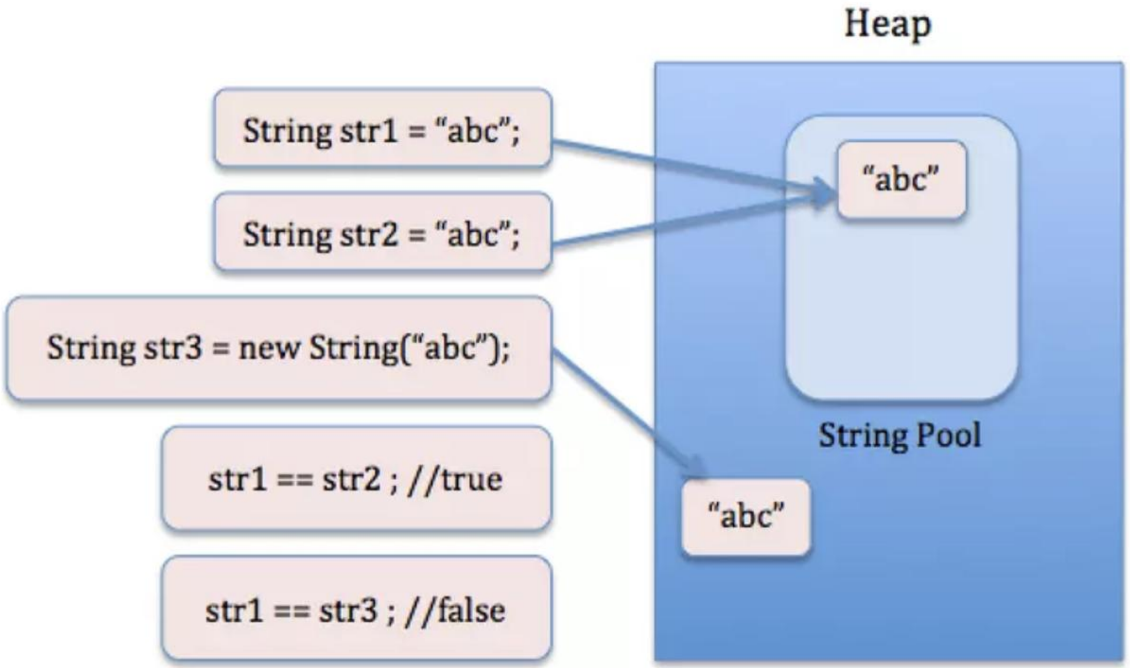
Tip: For heavy string manipulation, consider StringBuilder or StringBuffer (they are mutable).

Pitfall: Doing many concatenations in a loop can cause performance bottlenecks.

EQUALITY VS. IDENTITY IN STRINGS

- **==** checks reference identity (are they the same object?).
- **.equals(...)** checks actual character-by-character equality.

```
String str1 = "Hello";
String str2 = new String("Hello");
System.out.println(str1 == str2);      // false
System.out.println(str1.equals(str2)); // true
```



Remind that **all** object comparisons (besides primitives) typically use .equals(...), not ==.

Best Practice : Always use .equals() to compare strings

ENUM

What is an Enum?

An **enum (enumeration)** is a special Java type used to define **a fixed set of constants**.
Unlike regular constants (final static variables), enums are **type-safe** and can include **fields, methods, and constructors**.

They are often used to represent **a predefined set of values (**Days of the week , Order statuses , Traffic light colors)

```
enum UserRole {
    ADMIN(3),
    MODERATOR(2),
    USER(1);
    private final int permissionLevel;
    UserRole(int level) {
        this.permissionLevel = level;
    }
    public boolean canAccessAdminPanel() {
        return this.permissionLevel >= 3;}
}
```

```
public class EnumExample {
    public static void main(String[] args) {
        UserRole role = UserRole.MODERATOR;

        System.out.println("User Role: " + role);
        System.out.println("Can access admin panel? " + role.canAccessAdminPanel());
        role = UserRole.ADMIN;
        System.out.println("Can access admin panel now? " + role.canAccessAdminPanel());}
}
```

- QUESTIONS :Why are Enums preferred over constants (final static int) ?

ANSWER

Feature	Enums	final static int
Type Safety	✅ Prevents invalid values	❌ Can accept any integer
Readability	✅ Self-explanatory	❌ Hard to read (1, 2 instead of ADMIN, EDITOR)
Behavior	✅ Can have methods & fields	❌ Only stores values
Compilation Safety	✅ Compile-time checks	❌ No checks (wrong int can be passed)
Iteration Support	✅ Can use .values() to loop	❌ Manual array needed
Switch Compatibility	✅ Works natively	❌ Requires additional handling
Serialization & Reflection	✅ Works easily	❌ Extra steps required
Thread Safety	✅ Immutable & safe	❌ Needs extra work

WRITING JAVADOC – INTRODUCTION

Javadoc :A specialized tool that parses /** ... */ comments to produce HTML docs.

Purpose :

1)Standardize documentation.

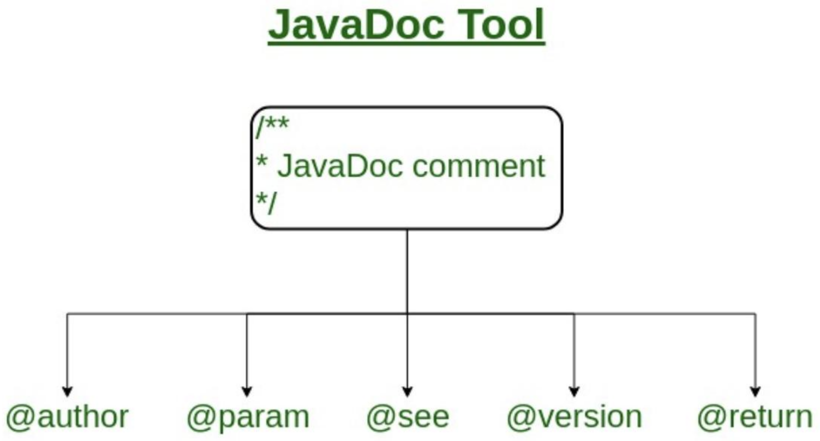
2) Provide quick references for other developers.

3) Make it easy to find usage examples.

Best Practice:Write Javadoc **while** coding, not after. Ensures docs stay up-to-date.

- ❖ **@author**: who wrote it
- ❖ **@version**: version of the class or method
- ❖ **@param**: describes method parameters
- ❖ **@return**: describes what the method returns
- ❖ **@throws** or @exception: which exceptions might be thrown
- ❖ **@see**: references other classes or methods

```
/**
 * Brief description (one sentence).
 * More details about this class or method.
 *
 * @author ...
 * @version ...
 * @param ...
 * @return ...
 * @throws ...
 */
```



JavaDoc Tool

/**
* JavaDoc comment
*/

@author @param @see @version @return

BUILT-IN PACKAGES – OVERVIEW

Why ?

Productivity: Save time by leveraging well-tested code.
Reliability:The standard library is robust and well-documented.

Syntax : `import java.util.ArrayList;`
WildCard : `import java.util.*;`

Avoid overshadowing classes from different packages with the same class name (e.g., java.sql.Date vs. java.util.Date).

LocalDateTime (java.time.LocalDateTime) – Modern Date and Time API
Random (java.util.Random) – Generating Random Numbers

LOCAL DATE TIME

Example: Formatting and Date Calculation

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class LocalDateTimeFormat {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter format = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String formattedDate = now.format(format);
        System.out.println("Formatted Date: " + formattedDate);
        // Add 10 days and subtract 2 hours
        LocalDateTime futureDate = now.plusDays(10).minusHours(2);
        System.out.println("Future Date: " + futureDate.format(format));
    }
}
```

Method	Description
now()	Returns the current date and time.
of(y, m, d, h, m, s)	Creates a specific date-time.
plusDays(n), minusDays(n)	Adds or subtracts days.
getDayOfWeek()	Returns the day of the week.
toLocalDate(), toLocalTime()	Extracts date or time only.

RANDOM

Used for:

- Simulating dice rolls, lotteries, or games.
- Generating unique test data.
- Shuffling items in a collection.

Method	Description
nextInt()	Returns a random int from Integer.MIN_VALUE to Integer.MAX_VALUE.
nextInt(bound)	Returns a random int between 0 and bound-1.
nextDouble()	Returns a random decimal between 0.0 and 1.0.
nextBoolean()	Returns a random true or false.

```
import java.util.Random;
public class RandomExample {
    public static void main(String[] args) {
        Random rand = new Random();
        System.out.println("Random int: " + rand.nextInt(100)); // 0-99
        System.out.println("Random double: " + rand.nextDouble()); // 0.0 - 1.0
    }
}
```

TIME TO CODE

- ***Hotel Reservation System*** with classes for Hotel, Room, Booking, Guest

  Hotel:
  Contains multiple Room objects (composition).
  Manages a list of Booking objects (aggregation).

  Room:
  Must reference an enum (RoomType) to specify the type .
  Tracks its own occupancy status and price.

  Booking:
  References one or more Room objects.
  Stores check-in and check-out dates using LocalDateTime.
  Maintains a Booking Status using an enum.
  Associates with a Guest (aggregation).

  Guest:
  Stores basic information such as name and contact details (String fields , etc)

پایان