Aggregation & Composition in Java

Lecture 07

Aggregation in Java

- **Exploring Object Relationships with Practical Java Code Examples**

  - **Aggregation: A form of association where one object contains another, but both can exist independently.**

  - **Key Concept: "Has-a" relationship between two classes.**

  - **Practical example demonstrating aggregation in Java.**

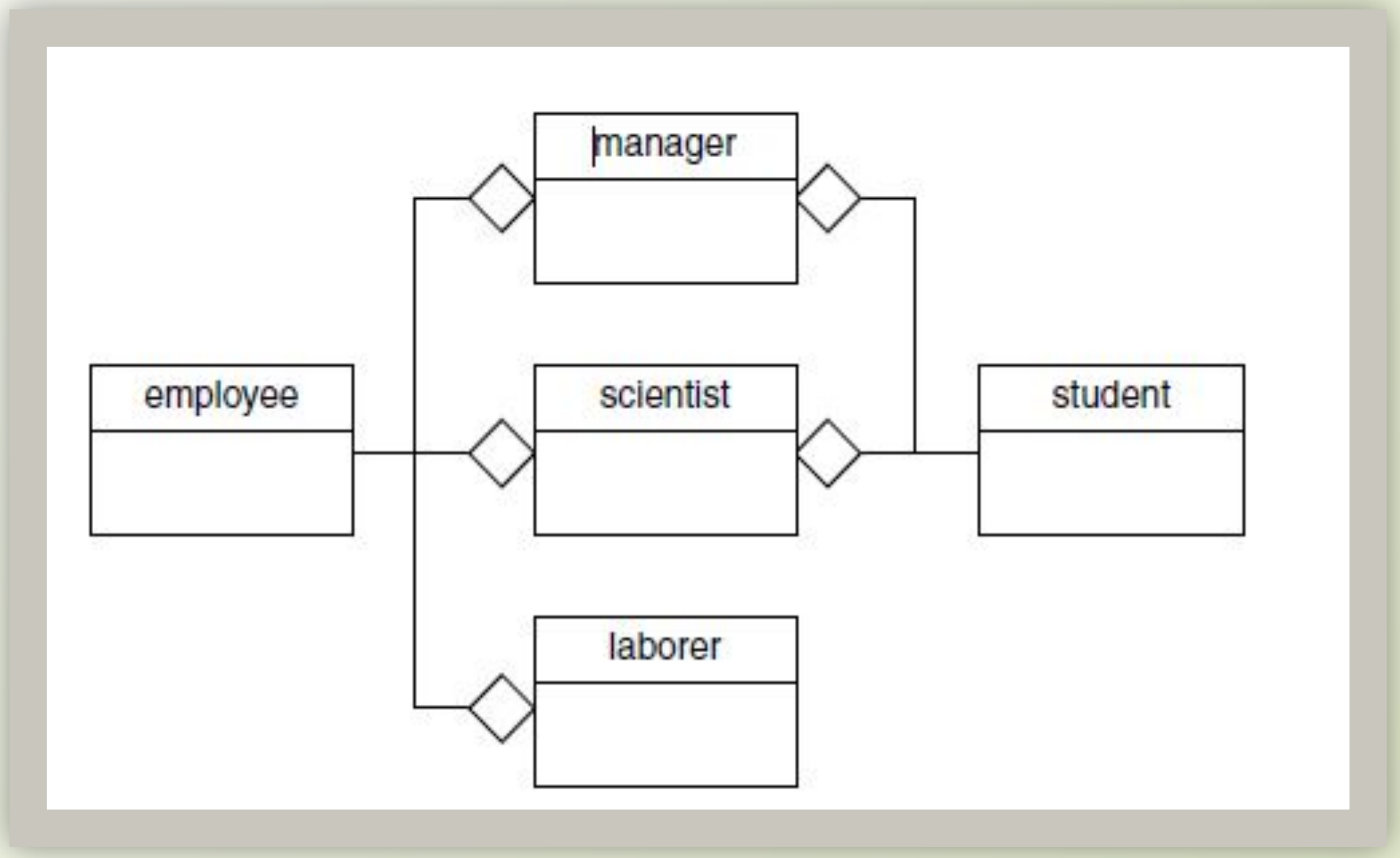
What is Aggregation?

- **Aggregation represents a relationship where one class holds a reference to another class, but both can function independently.**

- **Key Features:**
    - "Has-a" relationship.
    - Both objects have their own lifecycle.
    - Parent object can contain multiple instances of the child object.

What is Aggregation?

- If a class B is derived by inheritance from a class A, we can say that "B is a kind of A." This is because B has all the characteristics of A, and in addition some of its own.



What is Aggregation?

- Aggregation is called a “has a” relationship. We say a library has a book or an invoice has an item line

- In object-oriented programming, aggregation may occur when one object is an attribute of another. Here’s a case where an object of class A is an attribute of class B:

```
class A
{};
class B
{
A objA; // define objA as an object of class A
};
```

Real-world Example of Aggregation

- Example: A Library contains multiple Books, but Books can exist independently of the Library.

- Analogy: Just like books can be in multiple libraries, they still exist when not part of any library.

Aggregation Example: Code

```
class Book {
    String title;
    String author;

    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
}
```

Aggregation Example: Code

```
class Library {
    private List<Book> books;
    Library(List<Book> books) {
        this.books = books;
    }
    public void showBooks() {
        for (Book book : books) {
            System.out.println(book.title + " by " + book.author);
        }
    }
}
```

Aggregation Example: Code

```
public class AggregationExample {
    public static void main(String[] args) {
        Book book1 = new Book("1984", "George Orwell");
        Book book2 = new Book("To Kill a Mockingbird", "Harper Lee");

        List<Book> books = Arrays.asList(book1, book2);
        Library library = new Library(books);

        library.showBooks();
    }
}
```

Key Characteristics of Aggregation

Independence: Both classes can exist independently.

Ownership: The container class (like Library) does not fully "own" the contained class (Book).

Reusability: Aggregation allows for objects to be used by multiple other classes.

Advantages of Aggregation

Flexible Design: Aggregation allows independent lifecycles, making the design more flexible.

Reusability: Objects can be reused across multiple classes or contexts.

Reduced Coupling: Aggregated objects are loosely coupled, improving maintainability.

When to Use Aggregation?

When the child object can exist without the parent object.

Examples:

- A Team has Players, but Players can exist without a Team.
- A Company has Employees, but Employees can exist without the Company.

When to Use Aggregation?

When the child object can exist without the parent object.

Examples:

- A Team has Players, but Players can exist without a Team.
- A Company has Employees, but Employees can exist without the Company.

Introduction to Composition

What is Composition in Java?

- Composition is a design principle where a class is composed of one or more objects from other classes.
- It's a "Has-A" relationship (e.g., a car has an engine).
- Provides a way to reuse code without inheritance.
- More flexible than inheritance
- promotes loose coupling.

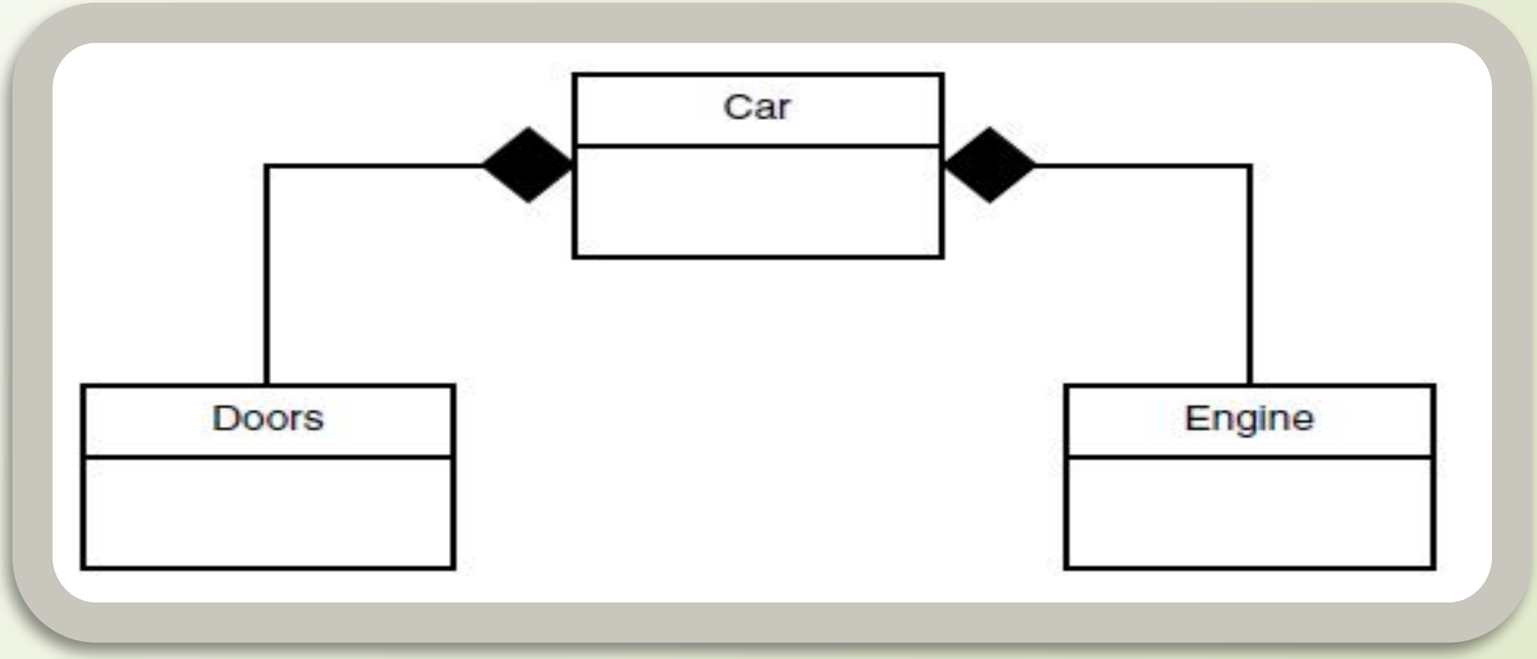
Introduction to Composition

- Composition is a stronger form of aggregation. It has all the characteristics of aggregation, plus two more:

  - The part may belong to only one whole.
  - The lifetime of the part is the same as the lifetime of the whole.
  - Child does not have its own life cycle
  - If parent object gets deleted then all of its child objects will be deleted

Introduction to Composition

- A car is composed of doors (among other things). The doors can’t belong to some other car, and they are born and die along with the car. A room is composed of a floor, ceiling, and walls. While aggregation is a “has a” relationship, composition is a “**consists of**” relationship.



Introduction to Composition

What is Composition in Java?

- Composition is a design principle where a class is composed of one or more objects from other classes.
- It's a "Has-A" relationship (e.g., a car has an engine).
- Provides a way to reuse code without inheritance.

Why Use Composition?

Benefits of Composition

Reusability: Code from other classes can be reused without needing inheritance.

Flexible Design: Classes can evolve independently.

Avoiding Inheritance Pitfalls: Avoids issues like the fragile base class problem.

Why Use Composition?

Benefits of Composition

Reusability: Code from other classes can be reused without needing inheritance.

Flexible Design: Classes can evolve independently.

Avoiding Inheritance Pitfalls: Avoids issues like the fragile base class problem.

Composition vs Inheritance

Composition	Inheritance
Has-A relationship	Is-A relationship
Flexible	Rigid hierarchy
Promotes loose coupling	Can lead to tight coupling
Better suited for dynamic relationships	Suited for a fixed hierarchy

Example 1 - Class with Composition

```
// Class for Engine
class Engine {
    public void start() {
        System.out.println("Engine started.");
    }
}
// Class for Car that uses Engine
class Car {
    private Engine engine; // Composition
    public Car() {
        this.engine = new Engine(); // Car has an Engine
    }
    public void startCar() {
        engine.start(); // Using Engine's start method
        System.out.println("Car is running.");
    }
}
```

Example 1 - Class with Composition

```
// Main class to test
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.startCar(); // Output: Engine started. Car is running.
    }
}
```

Example 2 - Library and Books

```
// Book Class
class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }
}
```

Example 2 - Library and Books

```
// Library Class using Composition
class Library {
    private List<Book> books; // Composition
    public Library() {
        books = new ArrayList<>();
    }
    public void addBook(Book book) {
        books.add(book);
    }
    public void showBooks() {
        for (Book book : books) {
            System.out.println(book.getTitle());
        }
    }
```

Example 2 - Library and Books

```
// Main Class to test
public class Main {
    public static void main(String[] args) {
        Library library = new Library();
        library.addBook(new Book("Java Programming"));
        library.addBook(new Book("Data Structures"));

        library.showBooks(); // Output: Java Programming, Data Structures
    }
}
```

When to Use Composition?

- You want to reuse existing functionality.
- You need flexibility in class relationships.
- You don’t want a rigid hierarchy (inheritance).
- You are following design principles like SOLID.

Key Takeaways

- **Composition promotes flexibility and loose coupling.**
- **It enables code reuse without inheritance.**
- **Ideal when designing complex systems with multiple objects interacting.**
- **Use when you need a Has-A relationship.**

Scenario-Based Question on Composition in Java

You are tasked with designing a Smart Home Automation System where different components (like lights, thermostat, and security system) work together. The system should have a central control system that manages these components.

The requirements are as follows:

- The system should manage multiple devices.
- Each device should have its own class (e.g., Light, Thermostat, SecuritySystem).
- The central controller should be able to turn all devices on or off at once.
- Use composition to ensure that the central controller can interact with multiple devices without inheriting from them.

Task:

- Design the classes to implement this Smart Home Automation System.
- Use composition to create a central controller that manages multiple devices.
- Write a Java program to demonstrate how the central controller can control the devices.

Scenario-Based Question on Composition in Java

```
// Light class
class Light {
    private boolean isOn;

    public void turnOn() {
        isOn = true;
        System.out.println("Light is turned on.");
    }

    public void turnOff() {
        isOn = false;
        System.out.println("Light is turned off.");
    }
}
```

Scenario-Based Question on Composition in Java

```
// Thermostat class
class Thermostat {
    private boolean isOn;

    public void turnOn() {
        isOn = true;
        System.out.println("Thermostat is turned on.");
    }

    public void turnOff() {
        isOn = false;
        System.out.println("Thermostat is turned off.");
    }
}
```

Scenario-Based Question on Composition in Java

```
// SecuritySystem class
class SecuritySystem {
    private boolean isActivated;

    public void activate() {
        isActivated = true;
        System.out.println("Security system is activated.");
    }

    public void deactivate() {
        isActivated = false;
        System.out.println("Security system is deactivated.");
    }
}
```

Scenario-Based Question on Composition in Java

```java
// CentralControl class using composition
class CentralControl {
    private Light light; // Composition
    private Thermostat thermostat; // Composition
    private SecuritySystem securitySystem; // Composition

    public CentralControl(Light light, Thermostat thermostat, SecuritySystem securitySystem) {
        this.light = light;
        this.thermostat = thermostat;
        this.securitySystem = securitySystem;
    }
```

Scenario-Based Question on Composition in Java

```
  public void turnAllOn() {
      light.turnOn();
      thermostat.turnOn();
      securitySystem.activate();
      System.out.println("All devices are now ON.");
   }
   public void turnAllOff() {
      light.turnOff();
      thermostat.turnOff();
      securitySystem.deactivate();
      System.out.println("All devices are now OFF.");
   }
}
```

Scenario-Based Question on Composition in Java

```
// Main class to test the functionality
public class Main {
    public static void main(String[] args) {
        // Create individual devices
        Light light = new Light();
        Thermostat thermostat = new Thermostat();
        SecuritySystem securitySystem = new SecuritySystem();

        // Create a central controller that manages these devices
        CentralControl centralControl = new CentralControl(light, thermostat, securitySystem);

        // Test turning all devices on
        centralControl.turnAllOn();

        // Test turning all devices off
        centralControl.turnAllOff();
    }
```