Aggregation

In the last chapter, we explored Association, the fundamental "uses-a" relationship that connects independent objects, like a doctor and their patients. We learned that in an association, objects have their own lifecycles.

But what happens when the relationship is a bit tighter? What if one class represents a "whole" and another represents a "part" of that whole?

Think of a university department and its professors, a team and its players, or a playlist and its songs.

This is where Aggregation comes in. It’s a specialized, stronger form of association that models a "whole-part" relationship.

1. What is Aggregation?
Aggregation represents a "has-a" relationship between two classes, where one class (the whole) groups or organizes other classes (the parts), but does not control their lifecycle.

Key Characteristics of Aggregation:
The whole and the part are logically connected.
The part can exist independently of the whole.
The whole does not own the part.
The part can be shared among multiple wholes.
Both the whole and the part can be created and destroyed independently.
If a class contains other classes for logical grouping only without lifecycle ownership, it is an aggregation.

Real-World Analogy
Let’s consider a university context:

A Department has many Professors.
Professors may belong to a department, but they are not owned by it.
If a department is dissolved, the professors still continue to exist, possibly getting reassigned to other departments.
A professor can even belong to multiple departments in some universities.
This relationship models Aggregation—the department and professors are linked, but their lifecycles are not tightly coupled.

2. Code Example
Let’s model the above example in code:

Python

```
class Professor:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name
```

Java

```
class Professor {
    private String name;

    public Professor(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Python

```


class Department:
    def __init__(self, name, professors):
        self.name = name
        self.professors = professors

    def print_professors(self):
        print(f"Professors in {self.name} Department:")
        for professor in self.professors:
            print(f"- {professor.get_name()}")
```

Usage:

Java

```
class Department {
    private String name;
    private List<Professor> professors;

    public Department(String name, List<Professor> professors) {
        this.name = name;
        this.professors = professors;
```

```
    }

    public void printProfessors() {
        System.out.println("Professors in " + name + " Department:");
        for (Professor professor : professors) {
            System.out.println("- " + professor.getName());
        }
    }
}
```

Python

```
if __name__ == "__main__":
    p1 = Professor("Dr. Smith")
    p2 = Professor("Dr. Johnson")

    profs = [p1, p2]

    cs_dept = Department("Computer Science", profs)
    cs_dept.print_professors()
```

Java

```
public class Main {
    public static void main(String[] args) {
        Professor p1 = new Professor("Dr. Smith");
        Professor p2 = new Professor("Dr. Johnson");

        List<Professor> profs = List.of(p1, p2);

        Department csDept = new Department("Computer Science", profs);
        csDept.printProfessors();

        // csDept can be deleted or go out of scope...
        // but p1 and p2 still exist and can be used elsewhere.
    }
}
```

```
    # cs_dept can go out of scope or be deleted...
    # but p1 and p2 still exist and can be used elsewhere.
```

Department groups Professor objects.
The professors are not created inside the Department class.
They can exist before, and survive after, the department’s existence.

If you delete the csDept object, the professors still exist in memory and could be reassigned to another department. That's aggregation in action.

Department groups Professor objects.
The professors are not created inside the Department class.
They can exist before, and survive after, the department's existence.
If you delete the csDept object, the professors still exist in memory and could be reassigned to another department. That's aggregation in action.

3. UML Representation
In UML class diagrams, aggregation is represented by a hollow diamond (◇) at the "whole" (container) side of the relationship.

Department ◇———→ Professor
This reads as: A Department has Professors, but it does not own them.

4. When to Use Aggregation in OOP
Use aggregation when:

The part can exist independently of the whole.
The whole groups or organizes the parts logically.
The part might be shared across multiple wholes.
There is no ownership or lifecycle dependency.