(/)

Composition, Aggregation, and Association in Java

Last modified: June 23, 2022

by Attila Fejér (https://www.baeldung.com/author/attila-fejer/)

Java (https://www.baeldung.com/category/java/) +

1. Introduction

Objects have relationships between them, both in real life and in programming. Sometimes it's difficult to understand or implement these relationships.

In this tutorial, we'll focus on Java's take on three sometimes easily mixed up types of relationships: composition, aggregation, and association.

2. Composition

Composition is a "belongs-to" type of relationship. It means that one of the objects is a logically larger structure, which contains the other object. In other words, it's part or member of the other object.

Alternatively, **we often call it a "has-a" relationship** (as opposed to an "is-a" relationship, which is inheritance (/java-inheritance)).

For example, a room belongs to a building, or in other words a building has a room. So basically, whether we call it "belongs-to" or "has-a" is only a matter of point of view.

Composition is a strong kind of "has-a" relationship because the containing object owns it. Therefore, **the objects' lifecycles are tied. It means that if we destroy the owner object, its members also will be destroyed with it.** For example, the room is destroyed with the building in our previous example.

Note that doesn't mean, that the containing object can't exist without any of its parts. For example, we can tear down all the walls inside a building, hence destroy the rooms. But the building will still exist.

In terms of cardinality, a containing object can have as many parts as we want. However, **all of the parts need to have exactly one container**.

2.1. UML

In UML, we indicate composition with the following symbol:

(/wp-content/uploads/2019/08/composition.png)
Note, that the diamond is at the containing object and is the base of the line, not an arrowhead. For the sake of clarity, we often draw the arrowhead too:

(/wp-content/uploads/2019/08/composition-arrow.png)
So, then, we can use this UML construct for our Building-Room example:

(/wp-content/uploads/2019/08/composition-example.png)

2.2. Source Code

In Java, we can model this with a non-static inner class:

```
class Building {
    class Room {}
}
```

Alternatively, we can declare that class in a method body as well. It doesn't matter if it's a named class, an anonymous class or a lambda:

```
class Building {
    Room createAnonymousRoom() {
        return new Room() {
            @Override
            void doInRoom() {}
        };
    }

    Room createInlineRoom() {
        class InlineRoom implements Room {
            @Override
            void doInRoom() {}
        }
        return new InlineRoom();
    }

    Room createLambdaRoom() {
        return () -> {};
    }

    interface Room {
        void doInRoom();
    }
}
```

Note, that it's essential, that our inner class should be non-static since it binds all of its instances to the containing class.

Usually, the containing object wants to access its members. Therefore, we

should store their references:

```
class Building {
    List<Room> rooms;
    class Room {}
}
```

Note, that all inner class objects store an implicit reference to their containing object. As a result, we don't need to store it manually to access it:

```
class Building {
    String address;

    class Room {
        String getBuildingAddress() {
            return Building.this.address;
        }
    }
}
```

3. Aggregation

Aggregation is also a “has-a” relationship. What distinguishes it from composition, that it doesn't involve owning. As a result, the lifecycles of the objects aren't tied: every one of them can exist independently of each other.

For example, a car and its wheels. **We can take off the wheels, and they'll still exist.** We can mount other (preexisting) wheels, or install these to another car and everything will work just fine.

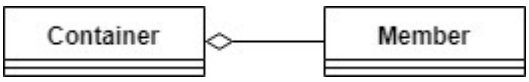
Of course, a car without wheels or a detached wheel won't be as useful as a car with its wheels on. But that's why this relationship existed in the first place: to **assemble the parts to a bigger construct, which is capable of more things than its parts**.

Since aggregation doesn't involve owning, **a member doesn't need to be tied to only one container**. For example, a triangle is made of segments. But triangles can share segments as their sides.

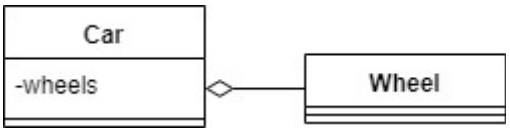
3.1. UML

Aggregation is very similar to composition. The only logical difference is aggregation is a weaker relationship.

Therefore, UML representations are also very similar. The only difference is the diamond is empty:



(/wp-content/uploads/2019/08/aggregation.png)
For cars and wheels, then, we'd do:



(/wp-content/uploads/2019/08/aggregation-example.png)

3.2. Source Code

In Java, we can model aggregation with a plain old reference:

```
class Wheel {}

class Car {
    List<Wheel> wheels;
}
```

The member can be any type of class, except a non-static inner class.

In the code snippet above both classes have their separate source file. However, we can also use a static inner class:

```
class Car {
    List<Wheel> wheels;
    static class Wheel {}
}
```

Note that Java will create an implicit reference only in non-static inner classes. Because of that, we have to maintain the relationship manually where we need it:

```
class Wheel {
    Car car;
}

class Car {
    List<Wheel> wheels;
}
```

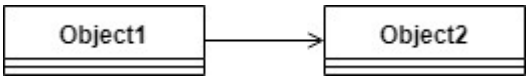
4. Association

Association is the weakest relationship between the three. **It isn't a "has-a" relationship**, none of the objects are parts or members of another.

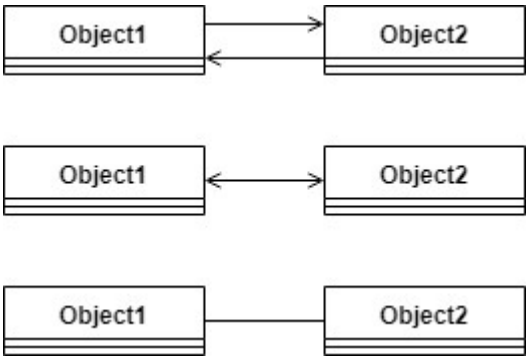
Association only means that the objects "know" each other. For example, a mother and her child.

4.1. UML

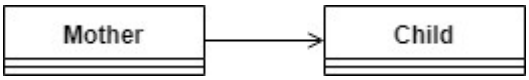
In UML, we can mark an association with an arrow:



(/wp-content/uploads/2019/08/association.png)
If the association is bidirectional, we can use two arrows, an arrow with an arrowhead on both ends, or a line without any arrowheads:



(/wp-content/uploads/2019/08/association-bidirectional.png)
We can represent a mother and her child in UML, then:



(/wp-content/uploads/2019/08/association-example.png)

4.2. Source Code

In Java, we can model association the same way as aggregation:

```
class Child {}

class Mother {
    List<Child> children;
}
```

But wait, **how can we tell if a reference means aggregation or association?**

Well, we can't. The difference is only logical: whether one of the objects is part of the other or not.

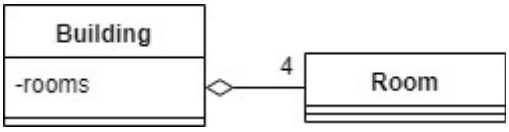
Also, we have to maintain the references manually on both ends as we did with aggregation:

```
class Child {
    Mother mother;
}

class Mother {
    List<Child> children;
}
```

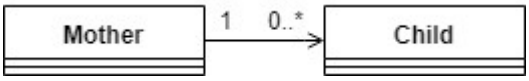
5. UML Sidenote

For the sake of clarity, sometimes we want to define the cardinality of a relationship on a UML diagram. We can do this by writing it to the ends of the arrow:



(/wp-content/uploads/2019/08/cardinality-1.png)
Note, that it doesn't make sense to write zero as cardinality, because it

means there's no relationship. The only exception is when we want to use a range to indicate an optional relationship:



(/wp-content/uploads/2019/08/cardinality-2.png)
Also note, that since in composition there's precisely one owner we don't indicate it on the diagrams.

6. A Complex Example

Let's see a (little) more complex example!

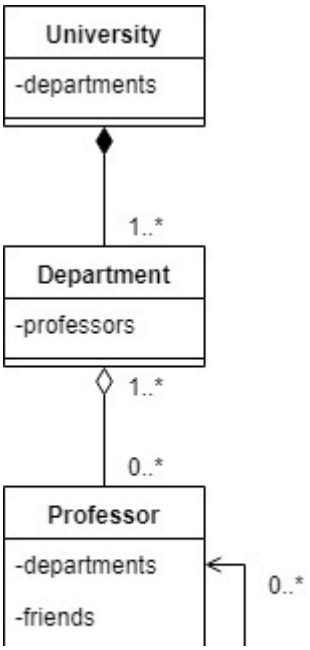
We'll model a university, which has its departments. Professors work in each department, who also has friends among each other.

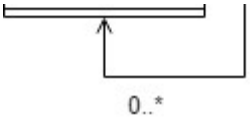
Will the departments exist after we close the university? Of course not, therefore it's a composition.

But the professors will still exist (hopefully). We have to decide which is more logical: if we consider professors as parts of the departments or not. Alternatively: are they members of the departments or not? Yes, they are. Hence it's an aggregation. On top of that, a professor can work in multiple departments.

The relationship between professors is association because it doesn't make any sense to say that a professor is part of another one.

As a result, we can model this example with the following UML diagram:





(/wp-content/uploads/2019/08/complex-example.png)
And the Java code looks like this:

```
class University {
    List<Department> department;
}

class Department {
    List<Professor> professors;
}

class Professor {
    List<Department> department;
    List<Professor> friends;
}
```

Note, that if we **rely on the terms "has-a", "belongs-to", "member-of", "part-of"**, and so on, we can more easily identify the relationships between our objects.

7. Conclusion

In this article, we saw the properties and representation of composition, aggregation, and association. We also saw how to model those relationships in UML and Java.

As usual, the examples are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-lang-oop-patterns).

Comments are closed on this article!

SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)