Object Oriented Programming

By Watcharin Sarachai

Topic

- Aggregation
- Composition
- UML
- Association

Aggregation

What is Aggregation in java?

- **Aggregation** is a special form of association. It is a relationship between two classes like association, however its a directional association is a one way association.

- If a class have an entity reference, it is known as **Aggregation.** It represents a HAS-A relationship.

- **For example:** two classes **Student** class and **Address** class. Student has an address so the **relationship** between student and address is a Has-A relationship.

- If you consider its vice versa, an **Address** doesn't need to have a **Student** necessarily.

- In **Aggregation**, both the entries can survive individually. (*ending one entity will not effect the other entity*).

```java
class Address {
  int streetNum;
  String city;
  String state;
  String country;

  Address(int street, String c, String st, String coun) {
    this.streetNum = street;
    this.city = c;
    this.state = st;
    this.country = coun;
  }
}
```

The above example :
In Student class, we declared a property of type **Address** to obtain student address.

```java
class StudentClass {
  int rollNum;
  String studentName;
  // Creating HAS-A relationship with Address class
  Address studentAddr;

  StudentClass(int roll, String name, Address addr) {
    this.rollNum = roll;
    this.studentName = name;
    this.studentAddr = addr;
  }

  public static void main(String args[]) {
    Address ad = new Address(55, "Agra", "UP", "India");
    StudentClass obj = new StudentClass(123, "Chaitanya", ad);
    System.out.println(obj.rollNum);
    System.out.println(obj.studentName);
    System.out.println(obj.studentAddr.streetNum);
    System.out.println(obj.studentAddr.city);
    System.out.println(obj.studentAddr.state);
    System.out.println(obj.studentAddr.country);
  }
}
```

```
Output:

123
Chaitanya
55
Agra
UP
India
```

What is Aggregation in java?

- **Employee** object contains many informations such as **id**, **name**, **address** etc.

  - It contains one more object named **address**, which contains its own informations such as **city**, **state**, **country**, **zipcode** etc. as given below.

```
class Employee {
  int id;
  String name;
  Address address;// Address is a class
  ...
}
```

```
class Address {
  String city;
  String state;
  String country;
  String zipcode;
  ...
}
```

Why we need Aggregation?

- To maintain code re-usability. To understand this lets take the example.
  - Suppose there are **College, Staff** classes and **Student**, **Address**. In order to maintain Student's **address**, **College Address** and **Staff's address** we don't need to use the same code again and again.
- We just have to use the reference of **Address** class while defining each of these classes like:

```
Student Has-A Address (Has-a relationship between student and address)
College Has-A Address (Has-a relationship between college and address)
Staff Has-A Address (Has-a relationship between staff and address)
```

Hence we can improve code re-usability by using Aggregation relationship.

If we write this in a program:

```
class Address {
  int streetNum;
  String city;
  String state;
  String country;

  Address(int street, String c, String st, String coun) {
    this.streetNum = street;
    this.city = c;
    this.state = st;
    this.country = coun;
  }
}

class StudentClass {
  int rollNum;
  String studentName;
  // Creating HAS-A relationship with Address class
  Address studentAddr;

  StudentClass(int roll, String name, Address addr) {
    this.rollNum = roll;
    this.studentName = name;
    this.studentAddr = addr;
  }
  ...
}
```

If write this in a program:

```
class College {
  String collegeName;
  // Creating HAS-A relationship with Address class
  Address collegeAddr;

  College(String name, Address addr) {
    this.collegeName = name;
    this.collegeAddr = addr;
  }
  ...
}

class Staff {
  String employeeName;
  // Creating HAS-A relationship with Address class
  Address employeeAddr;

  Staff(String name, Address addr) {
    this.employeeName = name;
    this.employeeAddr = addr;
  }
  ...
}
```

We didn't write the Address code in any of the three classes, we created the HAS-A relationship with the Address class to use the Address code.

Aggregation Example

- In example, **Employee** has an object of **Address**, address object contains its own informations such as **city**, **state**, **country** etc. In such case relationship is **Employee** HAS-A **address**.

```
public class Address {
  String city, state, country;

  public Address(String city, String state, String country) {
    this.city = city;
    this.state = state;
    this.country = country;
  }
}
```

```java
public class Emp {
  int id;
  String name;
  Address address;

  public Emp(int id, String name, Address address) {
    this.id = id;
    this.name = name;
    this.address = address;
  }
  void display() {
    System.out.println(id + " " + name);
    System.out.println(address.city + " " + address.state + " " + address.country);
  }
  public static void main(String[] args) {
    Address address1 = new Address("gzb", "UP", "india");
    Address address2 = new Address("gno", "UP", "india");
    Emp e = new Emp(111, "varun", address1);
    Emp e2 = new Emp(112, "arun", address2);
    e.display();
    e2.display();
  }
}
```

Aggregation Example

```java
// student class
class Student {
  String name;
  int id;
  String dept;

  Student(String name, int id, String dept) {

    this.name = name;
    this.id = id;
    this.dept = dept;
  }
}
```

Aggregation Example

```
/*
 * Department class contains list of student Objects. It is associated with
 * student class through its Object(s).
 */
class Department {

  String name;
  private List<Student> students;

  Department(String name, List<Student> students) {

    this.name = name;
    this.students = students;

  }

  public List<Student> getStudents() {
    return students;
  }
}
```

Aggregation Example

```
/*
 * Institute class contains list of Department Objects. It is asoociated with
 * Department class through its Object(s).
 */
class Institute {

  String instituteName;
  private List<Department> departments;

  Institute(String instituteName, List<Department> departments) {
    this.instituteName = instituteName;
    this.departments = departments;
  }

  // count total students of all departments
  // in a given institute
  public int getTotalStudentsInInstitute() {
    int noOfStudents = 0;
    List<Student> students;
    for (Department dept : departments) {
      students = dept.getStudents();
      for (Student s : students) {
        noOfStudents++;
      }
    }
    return noOfStudents;
  }
}
```

```java
// main method
class GFG {
  public static void main(String[] args) {
    Student s1 = new Student("Mia", 1, "CSE");
    Student s2 = new Student("Priya", 2, "CSE");
    Student s3 = new Student("John", 1, "EE");
    Student s4 = new Student("Rahul", 2, "EE");

    // making a List of
    // CSE Students.
    List<Student> cse_students = new ArrayList<Student>();
    cse_students.add(s1);
    cse_students.add(s2);

    // making a List of
    // EE Students
    List<Student> ee_students = new ArrayList<Student>();
    ee_students.add(s3);
    ee_students.add(s4);

    Department CSE = new Department("CSE", cse_students);
    Department EE = new Department("EE", ee_students);

    List<Department> departments = new ArrayList<Department>();
    departments.add(CSE);
    departments.add(EE);

    // creating an instance of Institute.
    Institute institute = new Institute("BITS", departments);

    System.out.print("Total students in institute: ");
    System.out.print(institute.getTotalStudentsInInstitute());
  }
}
```

```
Output:

Total students in institute: 4
```

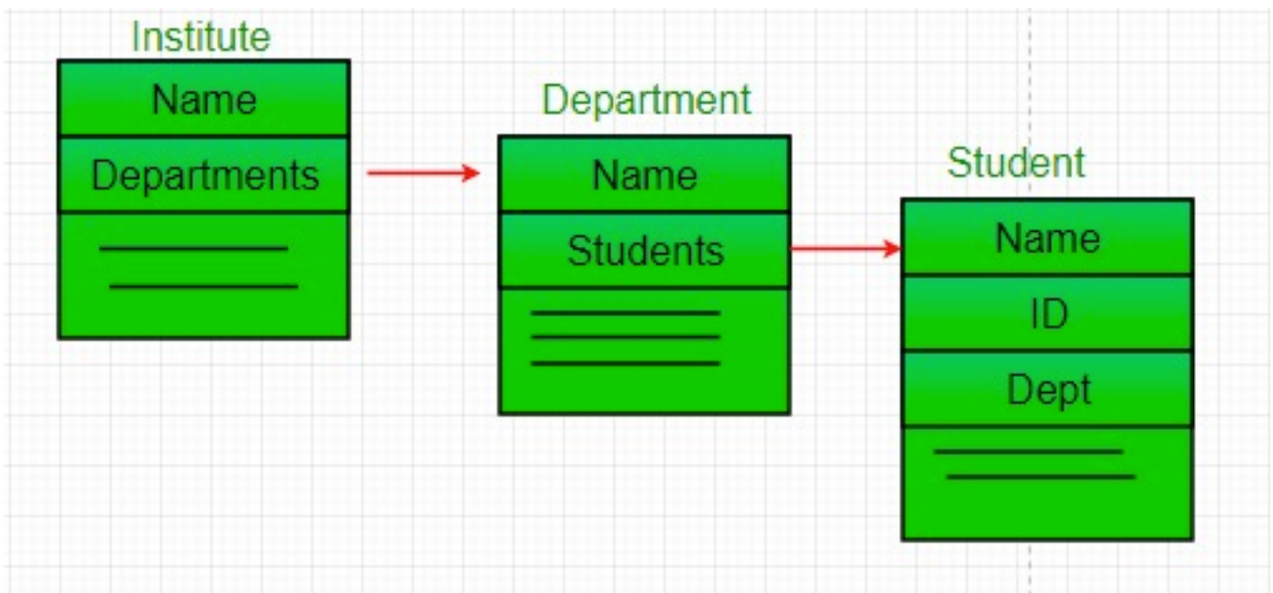
Aggregation

- In the previous example, there is an Institute which has number of departments. Every department has number of students.

- The Institute class has a reference to Object or number of Objects (i.e. List of Objects) of the Department class.

- That means Institute class is associated with Department class through its Object(s). And Department class has a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).

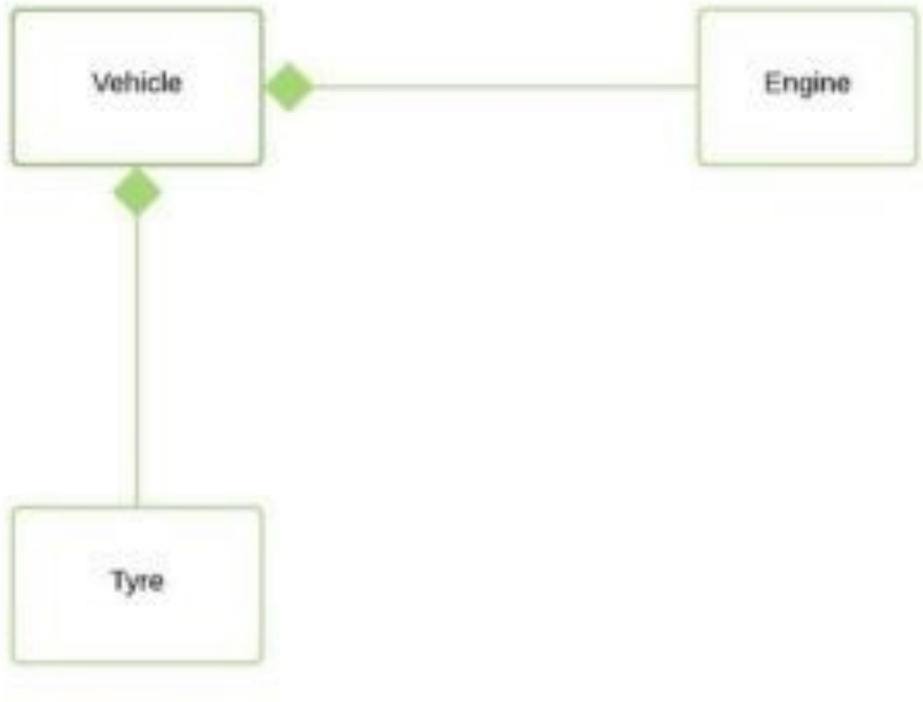
- It represents a Has-A relationship.



Composition

Composition

- Composition is restricted than Aggregation in which two entities are highly dependent on each other.
    - It represents part-of relationship.
- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object cannot exist without the other entity.



Composition

```java
// class book
class Book {

  public String title;
  public String author;

  Book(String title, String author) {

    this.title = title;
    this.author = author;
  }
}
```

```java
// Libary class contains
// list of books.
class Library {

  // reference to refer to list of books.
  private final List<Book> books;

  Library(List<Book> books) {
    this.books = books;
  }

  public List<Book> getTotalBooksInLibrary() {

    return books;
  }

}
```

Composition

```
// main method
class GFG {
  public static void main(String[] args) {

    // Creating the Objects of Book class.
    Book b1 = new Book("EffectiveJ Java", "Joshua Bloch");
    Book b2 = new Book("Thinking in Java", "Bruce Eckel");
    Book b3 = new Book("Java: The Complete Reference", "Herbert Schildt");

    // Creating the list which contains the
    // no. of books.
    List<Book> books = new ArrayList<Book>();
    books.add(b1);
    books.add(b2);
    books.add(b3);

    Library library = new Library(books);

    List<Book> bks = library.getTotalBooksInLibrary();
    for (Book bk : bks) {

      System.out.println("Title : " + bk.title + " and " + " Author : " + bk.author);
    }
  }
}
```

Composition

```
Output

Title : EffectiveJ Java and  Author : Joshua Bloch
Title : Thinking in Java and  Author : Bruce Eckel
Title : Java: The Complete Reference and  Author : Herbert Schildt
```

- In example a library can have number of books on same or different subjects.

- If the library gets destroyed then all books within that particular library will be destroyed. (book can not exist without library). That’s why it is composition.

Aggregation vs Composition

1. **Dependency:** Aggregation implies a relationship where the child can exist independently of the parent.

    - **For example**, **Bank** and **Employee**, delete the **Bank** and the **Employee** still exist.

2. **Composition** implies a relationship where the child cannot exist independent of the parent.

    - **Example: Human** and **heart**, **heart** don't exist separate to a **Human**

3. **Type of Relationship:** Aggregation relation is "has-a" and composition is "part-of" relation.

4. **Type of association:** Composition is a **strong Association** whereas Aggregation is a **weak Association**.

Aggregation vs Composition

```java
// Engine class which will
// be used by car. so 'Car'
// class will have a field
// of Engine type.
class Engine {
  // starting an engine.
  public void work() {

    System.out.println("Engine of car
    has been started ");

  }

}
```

```java
// Engine class which will
// Engine class
final class Car {

  // For a car to move,
  // it need to have a engine.
  private final Engine engine; // Composition
  // private Engine engine; // Aggregation

  Car(Engine engine) {
    this.engine = engine;
  }

  // car start moving by starting engine
  public void move() {

    // if(engine != null)
    {
      engine.work();
      System.out.println("Car is moving ");
    }
  }
}
```

Aggregation vs Composition

```
class GFG {
  public static void main(String[] args) {

    // making an engine by creating
    // an instance of Engine class.
    Engine engine = new Engine();

    // Making a car with engine.
    // so we are passing a engine
    // instance as an argument while
    // creating instace of Car.
    Car car = new Car(engine);
    car.move();

  }
}
```

```
Output:

Engine of car has been started
Car is moving
```

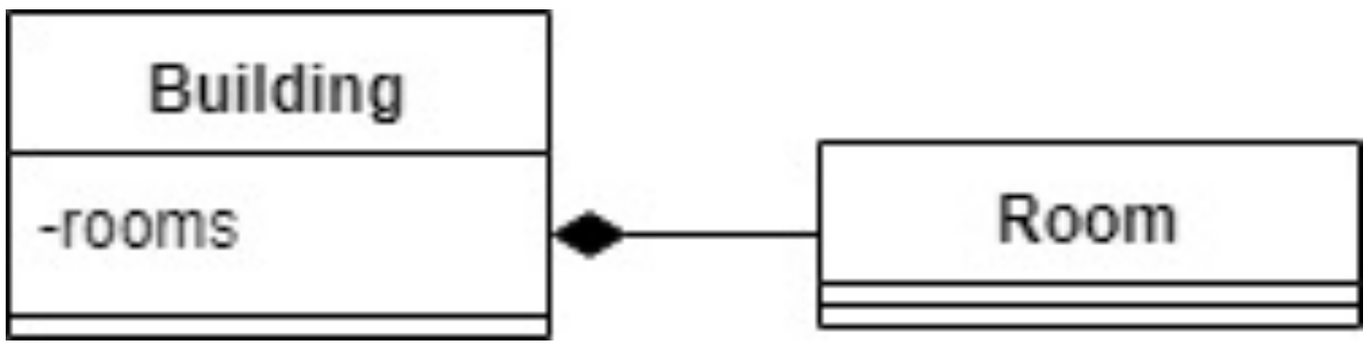
UML-Unified Modeling Language

- **Composition**
  - In UML, we indicate **composition** with the following symbol:

  - The diamond is at the containing object and is the base of the line, not an arrowhead. For clarity, we often draw the arrowhead too:

UML-Unified Modeling Language

- **Composition**
  - We can use this UML construct for our Building-Room example:

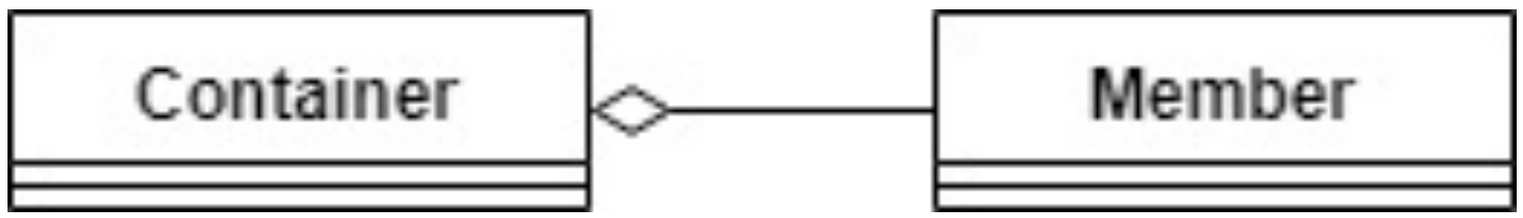


UML-Unified Modeling Language

- **Aggregation**

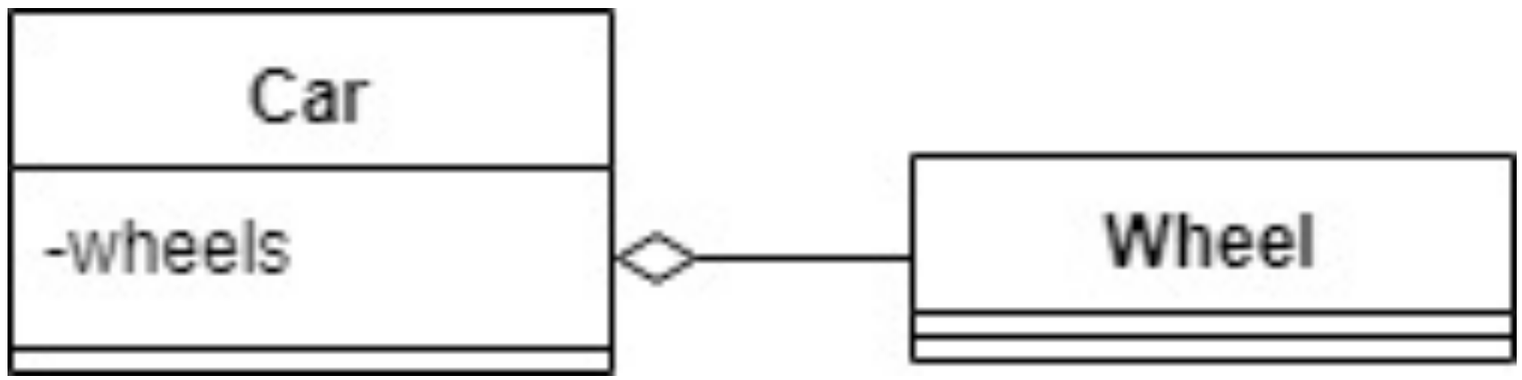
  - Aggregation is very similar to composition. The only logical difference is aggregation is a weaker relationship.

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



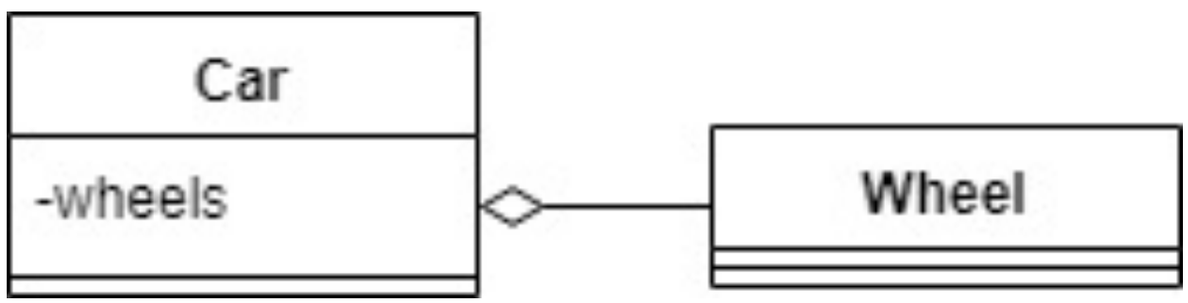
UML-Unified Modeling Language

- Aggregation
  - For cars and wheels, then, we'd do:



UML-Unified Modeling Language

- Aggregation
  - For cars and wheels, then, we'd do:

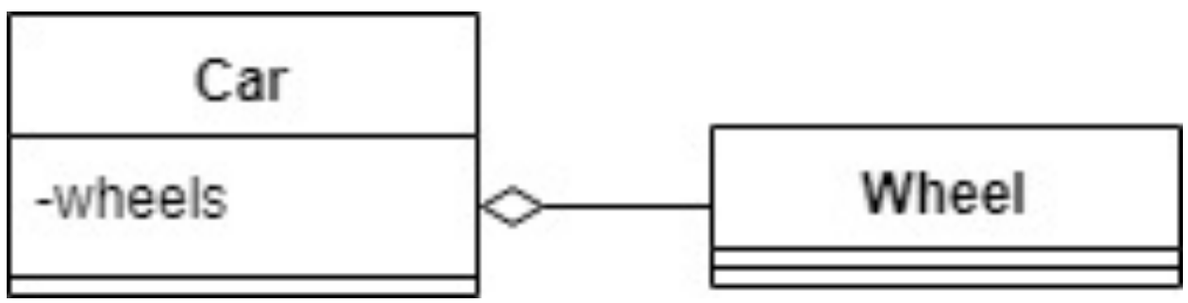


- Source Code

```
class Wheel {
}

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

UML-Unified Modeling Language

- Aggregation
  - For cars and wheels, then, we'd do:



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

```
class Wheel {
  Car car;
}

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

Association

What is Association in java?

- Association is relation between two separate classes which establishes through their Objects. Association can be **one-to-one**, **one-to-many**, **many-to-one**, **many-to-many**.

- In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.

- **Association** 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.

What is Association in java?

```java
// class bank
class Bank {
  private String name;

  // bank name
  Bank(String name) {
    this.name = name;
  }

  public String getBankName() {
    return this.name;
  }
}

// employee class
class Employee {
  private String name;

  // employee name
  Employee(String name) {
    this.name = name;
  }

  public String getEmployeeName() {
    return this.name;
  }
}
```

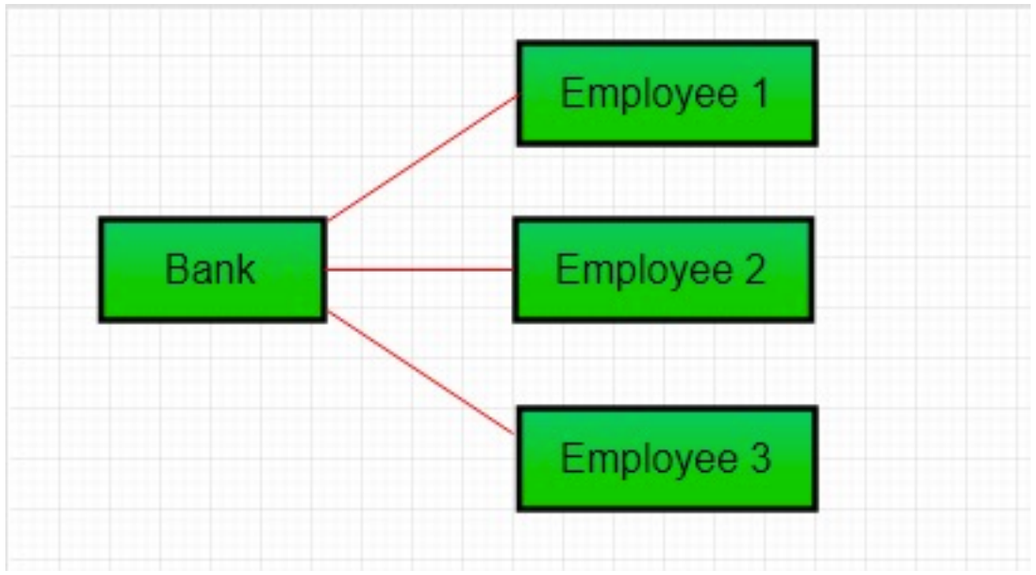
What is Association in java?

```
// Association between both the
// classes in main method
class Association {
  public static void main(String[] args) {
    Bank bank = new Bank("Axis");
    Employee emp = new Employee("Neha");

    System.out.println(emp.getEmployeeName() + " is employee of " + bank.getBankName());
  }
}
```

```
Output:

Neha is employee of Axis
```

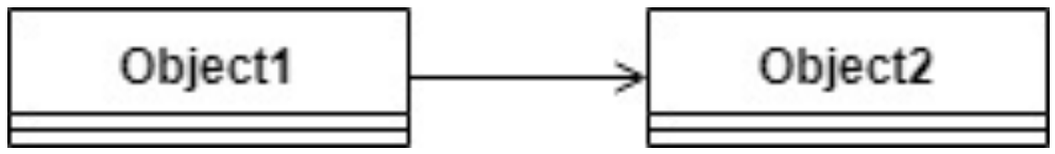
Association

- In above example two separate classes **Bank** and **Employee** are **associated** through their Objects. **Bank** can have many **employees**, So it is a **one-to-many** relationship.

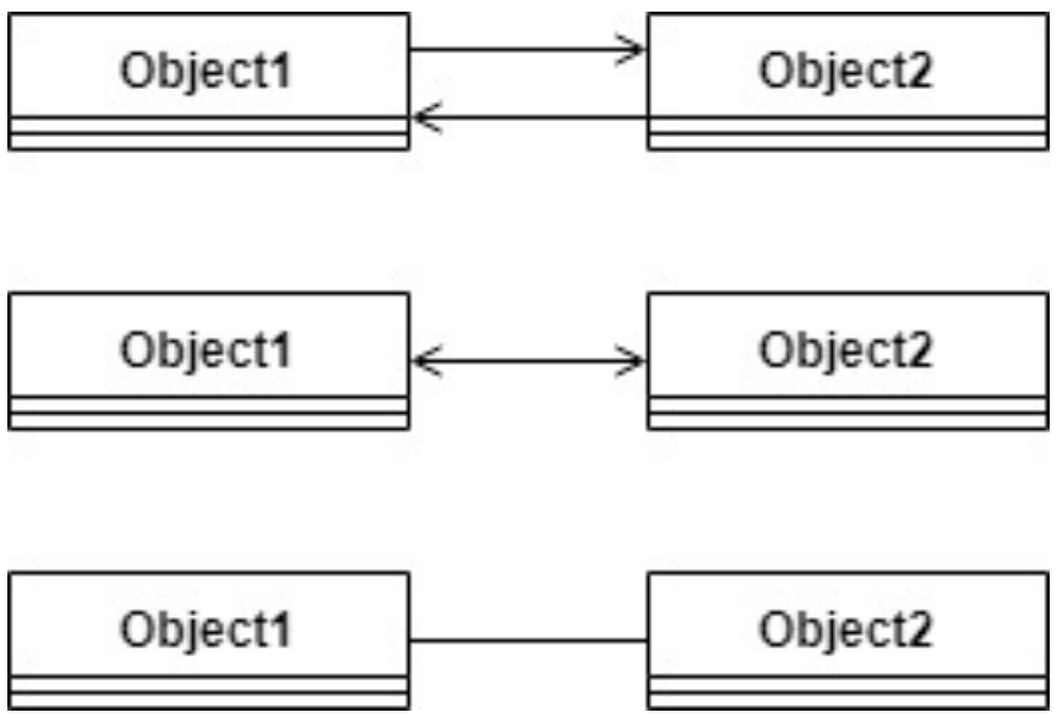


Association

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

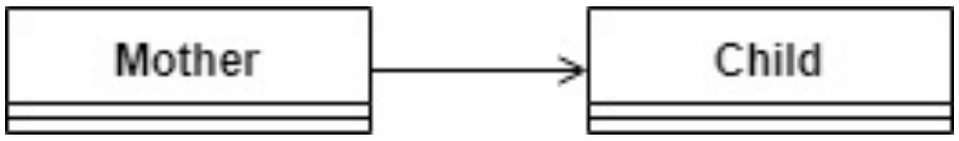


  - If the association is bidirectional, we can use two arrows, an arrow with an arrowhead on both ends, or a line without any arrowheads:



Association

- UML
  - We can represent a mother and her child in UML, then:

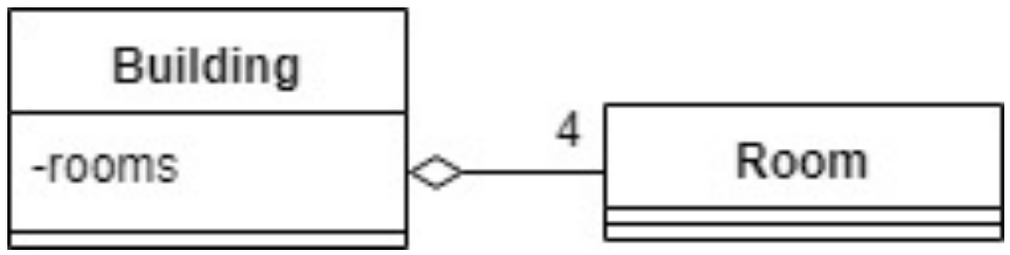


- Source Code

```
class Child {
}

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

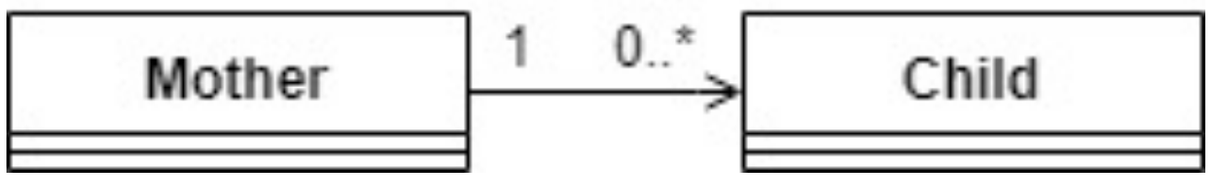
Association

- 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:



Association

- UML Sidenote
  - 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:



Also note, that since in composition there's precisely one owner we don't indicate it on the diagrams.

A Complex Example

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

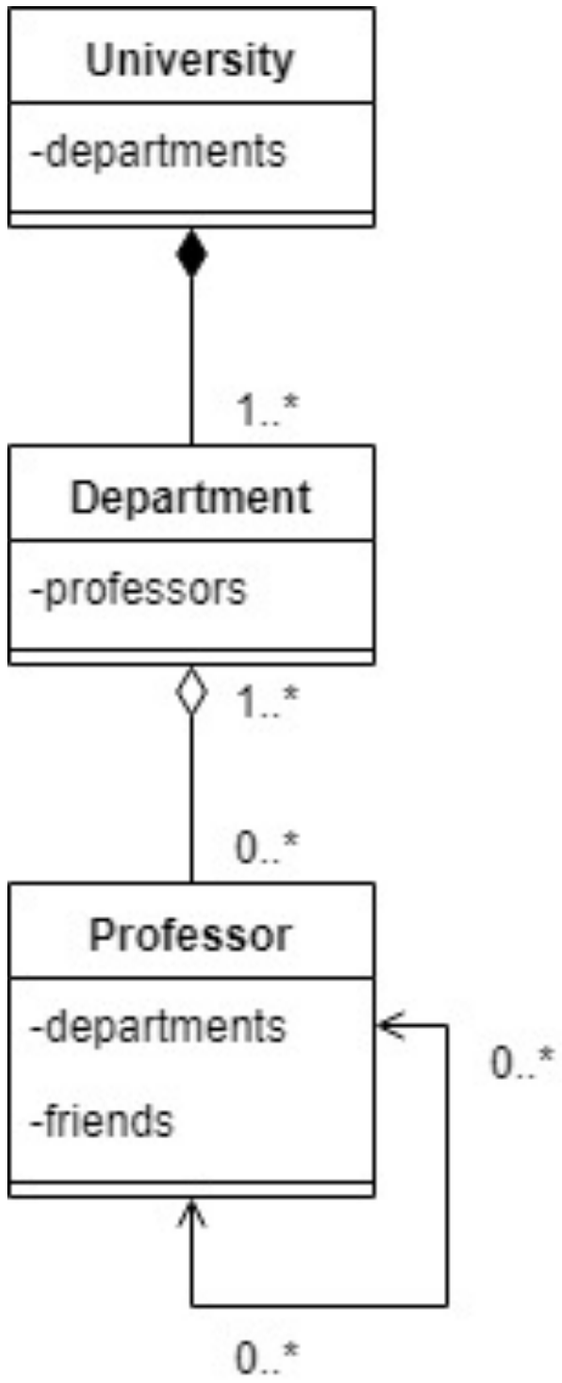
- Will the departments exist after we close the university? Of course not, therefore it's a **composition**.

- 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:

A Complex Example




```
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.

Reference

- **Super keyword in java with examplehttps://beginnersbook.com/2014/07/super-keyword-in-java-with-example/**
  Accessed: 2020-07-07
- **Association, Composition and Aggregation in Java - GeeksforGeekshttps://www.geeksforgeeks.org/association-composition-aggregation-java/**
  Accessed: 2020-07-09
- **Composition, Aggregation, and Association in Java | Baeldunghttps://www.baeldung.com/java-composition-aggregation-association**
  Accessed: 2020-07-09