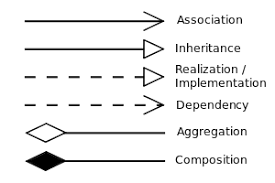
**Relationship types**

* Relationships between classes
  + Association
    - Semantic relation between classes
  + Inheritance
    - A class can inherit one or more classes
  + Aggregation
    - An association shows a class is a part of another class
  + Composition
    - A strong form of aggregation
  + Dependency
    - shows the dependency between classes



* **Aggregation:**
* An aggregation is a form of association that expresses a stronger (than normal association) coupling between class
* An aggregation is used between two classes
  + master and slave: “belongs to”
  + whole and part: “is a part of”

In a class diagram, an aggregation relationship can be illustrated with a "Library" class and a "Book" class. Here’s how it can be represented in Java

|  |
| --- |
| **<class> Library** |
| * Books:List<Book> |
| - name: String |
| + addBook(b: Book): void |
| + getBooks(): List<Book> |

|  |
| --- |
| **<class>** Book |
| - title: String |
| - author: String |
| + getDetails(): String |

**Java Code Example:**

import java.util.ArrayList;

import java.util.List;

**class Book** {

private String title;

private String author;

public Book(String title, String author) {

this.title = title;

this.author = author;

}

public String getDetails() {

return title + " by " + author;

}

}

**class Library** {

private String name;

private List<Book> books;

public Library(String name)

{

this.name = name;

this.books = new ArrayList<>();

}

public void addBook(**Book book**) {

books.add(book);

}

public List<Book> getBooks() {

return books;

}

}

### **Explanation:**

* The **Library** class aggregates **Book** instances.
* The aggregation indicates that while a **Library** can have multiple **Books**, the **Books** can exist independently of the **Library**.
* **Composition:**
* A composition is a strong form of aggregation
* A composition is also a “whole-part” relationship but the aggregate is stronger
  + If the whole is destroyed then parts will be also destroyed

A composition relationship can be illustrated with a "House" class and a "Room" class. Here’s how it can be represented in a class diagram.

|  |
| --- |
| **<class>** House |
| - address: String |
| + addRoom(r: Room): void |
| + getRooms(): List<Room> |

|  |
| --- |
| <class> Room |
| - name: String |
| + getName(): String |

**Java Code Example:**

import java.util.ArrayList;

import java.util.List;

**class Room** {

private String name;

public Room(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**class House** {

private String address;

private List<Room> rooms;

public House(String address) {

this.address = address;

this.rooms = new ArrayList<>();

}

public void addRoom(String roomName) {

rooms.add(new Room(roomName));

}

public List<Room> getRooms() {

return rooms;

}

}

### **Explanation:**

* The **House** class is composed of **Room** instances.
* In this composition relationship, **Rooms** cannot exist without a **House**; if the **House** is destroyed, the **Rooms** are also destroyed, emphasizing their strong lifecycle dependency.
* **Dependency: A class may depend on another class**
* **The dependency between classes can be implemented in different ways**
  + **Having an attribute with the type of another class**
  + **Sending a message using an attribute, a local variable, a global variable of another class or static methods**
  + **Receiving a parameter having type of another class**

A dependency relationship can be illustrated with a "Car" class that depends on an "Engine" class. Here’s how it can be represented in a class diagram:

|  |
| --- |
| **<class> Car** |
| - model: String |
| + startEngine(e: Engine): void |

|  |
| --- |
| **<class> Engine** |
| - horsepower: int |
| + getHorsepower(): int |

**Java Code Example:**

**class Engine** {

private int horsepower;

public Engine(int horsepower) {

this.horsepower = horsepower;

}

public int getHorsepower() {

return horsepower;

}

}

**class Car** {

private String model;

public Car(String model) {

this.model = model;

}

public void startEngine(**Engine engine**) {

System.out.println("Starting the engine with " + engine.getHorsepower() + " horsepower.");

}

}

### Explanation:

* The **Car** class has a dependency on the **Engine** class.
* This means that the **Car** uses an **Engine** to perform its functionality (e.g., starting the engine), but it doesn’t own the **Engine**. If the **Engine** changes, it may affect the **Car**, but the **Car** itself can exist independently of any specific **Engine** instance
* **Association**
* An association
  + is used to show how two classes are linked together
  + expresses a bidirectional semantic connection between classes
  + is an abstraction of the links between instances of classes

An association relationship can be illustrated with a "Teacher" class and a "Student" class, where a teacher can teach multiple students, and each student can have multiple teachers. Here’s how it can be represented in a class diagram:

|  |
| --- |
| **<class>** Teacher |
| - name: String |
| + addStudent(s: Student): void |
| + getStudents(): List<Student> |

**1**

**\***

|  |
| --- |
| **<class>** Student |
| - name: String |
| + getName(): String |

**Java Code Example:**

import java.util.ArrayList;

import java.util.List;

**class Student** {

private String name;

public Student(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**class Teacher** {

private String name;

private List<Student> students;

public Teacher(String name)

{

this.name = name;

this.students = new ArrayList<>();

}

public void addStudent(Student student) {

students.add(student);

}

public List<Student> getStudents() {

return students;

}

}

### Explanation:

* The **Teacher** class has an association with the **Student** class.
* A **Teacher** can have multiple **Students**, and each **Student** can be associated with multiple **Teachers**. This is a bidirectional association, as both classes can reference each other.
* The association is represented with a line in the class diagram, and the multiplicity indicates that one **Teacher** can be associated with many **Students** (1 to \*) and vice versa.
* **Inheritance:** A class can have several sub-classes

An inheritance relationship can be illustrated with a "Vehicle" class as a superclass and a "Car" class as a subclass. Here’s how it can be represented in a class diagram:

|  |
| --- |
| **<class>** Vehicle |
| - speed: int |
| + move(): void |
| + stop(): void |

|  |
| --- |
| **<class>** Car |
| - model: String |
| + honk(): void |

**Java Code Example:**

**class Vehicle** {

private int speed;

public int getSpeed()

{

Return this.speed;

}

public Vehicle(int speed) {

this.speed = speed;

}

public void move() {

System.out.println("The vehicle is moving at " + speed + " km/h.");

}

public void stop() {

System.out.println("The vehicle has stopped.");

}

}

**class Car extends Vehicle** {

private String model;

public Car(int speed, String model) {

super(speed);// Call the constructor of the superclass

this.model = model;

}

public void honk() {

System.out.println("The " + model + " goes honk!");

}

}

### Explanation:

* The **Vehicle** class is the superclass, containing common properties and methods for all vehicles, such as speed, move(), and stop().
* The **Car** class is a subclass that inherits from **Vehicle**. It adds its own property (model) and a method (honk()).
* Inheritance allows the **Car** class to use the methods and attributes of the **Vehicle** class, promoting code reuse and establishing a hierarchical relationship.