Association

- The association relationship expresses a semantic connection between classes:
  - There is no hierarchy.
- It is a relationship where two classes are weakly connected; i.e. they are merely “associates.”
  - All object have their own lifecycle;
  - There is no ownership.
  - There is no whole-part relationship.
- In another words, if inheritance and aggregation/composition doesn’t apply, it is most likely a simple association

Types of Association

- One-to-One
- One-to-Many
- Many-to-Many

Classification

- **Unidirectional Association**
- **Bidirectional Association**

Unidirectional Association

```
class Student {
public:
    std::string name;
    int grade;

    Student(std::string name, int grade) : name(name), grade(grade) {}

    void showGrade() {
        std::cout << name << "'s grade is: " << grade << std::endl;
    }
};
```

```
class Teacher {
private:
    Student* student;  // Teacher knows about Student
public:
    Teacher(Student* s) : student(s) {}

    void checkStudent() {
        std::cout << "Teacher is checking the student's details:"
<< std::endl;
        student->showGrade();
    }
};
```

```
int main() {
    Student s("Alice", 90);
    Teacher t(&s);  // Teacher knows about Student

    t.checkStudent();  // Teacher interacts with the Student

    return 0;
}
```

Bidirectional Association

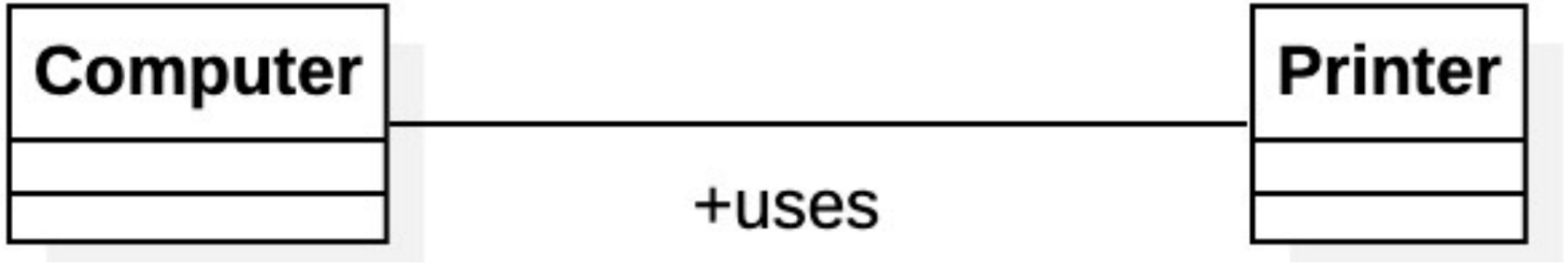
```
class Patient {
private:
   Doctor* doctor;  // Patient knows about Doctor
public:
   std::string name;

   Patient(std::string name) : name(name), doctor(nullptr) {}

   void setDoctor(Doctor* doc) {
      doctor = doc;
   }

   void showDoctor();
};
```

```
class Doctor {
private:
   Patient* patient;  // Doctor knows about Patient
public:
   std::string name;

   Doctor(std::string name) : name(name), patient(nullptr) {}

   void setPatient(Patient* pat) {
      patient = pat;
   }

   void showPatient() {
      std::cout << "Doctor " << name << " is treating patient: " <<
patient->name << std::endl;
   }
};

void Patient::showDoctor() {
   std::cout << "Patient " << name << " is being treated by doctor: "
<< doctor->name << std::endl;
}
```

```
int main() {
   Doctor doc("Dr. Smith");
   Patient pat("John Doe");
```

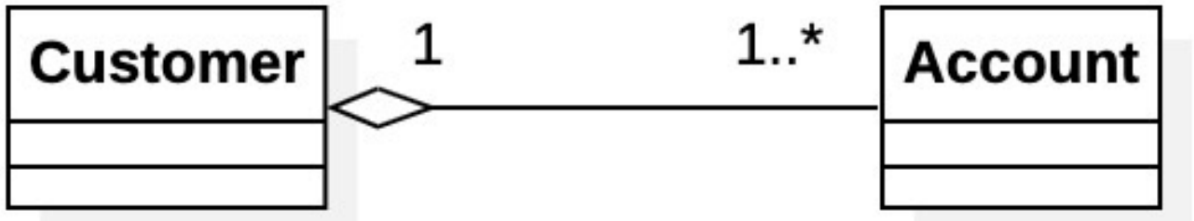
Association (Review)

- The association of two classes must be labeled.
- To improve the readability of diagrams, associations may be labeled in active or passive voice.



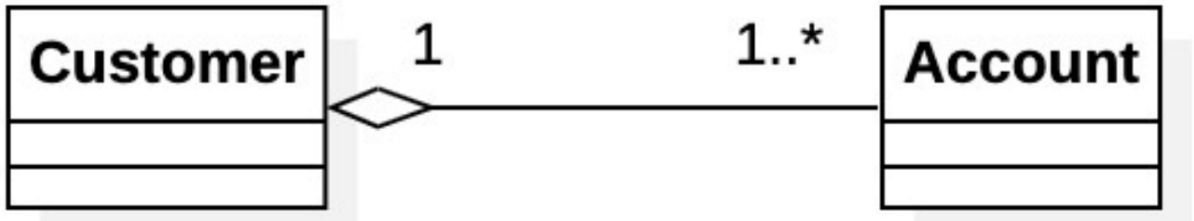
Aggregation (Review)

- An aggregation represents an asymmetric association, in which one of the ends plays a more important role than the other one.
- The following criteria imply an aggregation:
  - A class is part of another
  - The objects of one class are subordinates of the objects of another class
- Denotes a whole/part hierarchy with the ability to navigate from whole (aggregate) to its parts (attributes).
- The part is normally being referenced to either a pointer or by a reference in C++



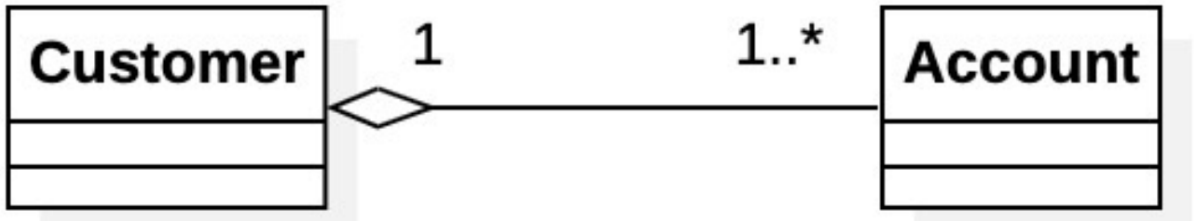
Aggregation (Review)

- An aggregation represents an asymmetric association, in which one of the ends plays a more important role than the other one.
- The following criteria imply an aggregation:
  - A class is part of another
  - The objects of one class are subordinates of the objects of another class
- Denotes a whole/part hierarchy with the ability to navigate from whole (aggregate) to its parts (attributes).
- The part is normally being referenced to either a pointer or by a reference in C++



Aggregation (Review)

- An aggregation represents an asymmetric association, in which one of the ends plays a more important role than the other one.
- The following criteria imply an aggregation:
  - A class is part of another
  - The objects of one class are subordinates of the objects of another class
- Denotes a whole/part hierarchy with the ability to navigate from whole (aggregate) to its parts (attributes).
- The part is normally being referenced to either a pointer or by a reference in C++



```
class Driver {
private:
    std::string driverName;

public:
    // Constructor to initialize the driver's name
    Driver(std::string name) : driverName(name) {}

    // Method to get the driver's name
    std::string getDriverName() const { return driverName; }
};
```

```
class Car {
private:
    std::string carModel;
    Driver* driver; // Aggregation: Car has a reference to a Driver

public:
    // Constructor to initialize the car model and assign a driver
    Car(std::string model, Driver* assignedDriver)
        : carModel(model), driver(assignedDriver) {}

    // Method to display car and driver information
    void display() const {
        std::cout << "Car Model: " << carModel
                  << ", Driven by: " << driver->getDriverName() << std::endl;
    }
};
```

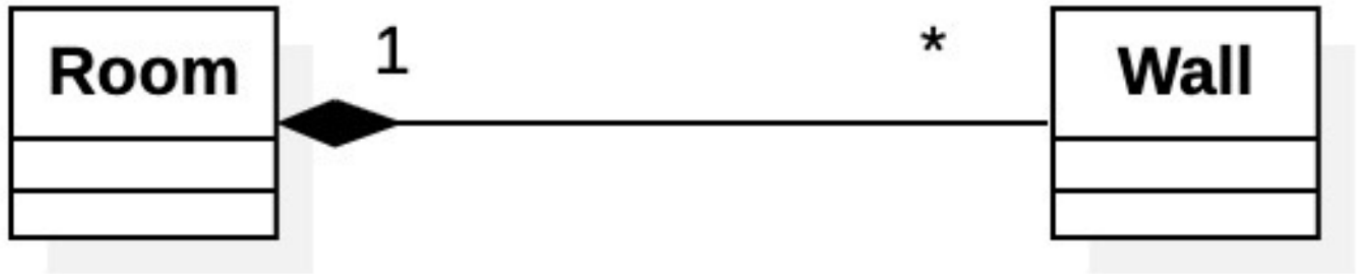
```
int main() {
    // Create a Driver object
    Driver john("John Doe");

    // Create a Car object and associate it with the Driver (aggregation)
    Car myCar("Toyota Corolla", &john);

    // Display information about the car and the driver
    myCar.display();

    return 0;
}
```

Key Features of Aggregation

1. **Weak Relationship**
2. **Ownership.**
3. **Implemented Using Pointers or References**

Aggregation (Review)

- A strong type of aggregation, when deletion of the whole causes the deletion of the part, is called ***composition.***
- The “part” objects are usually created in the constructor of the container class.



Key Features of Composition

- **Strong Relationship**
- **Ownership**
- **Lifetime Dependency**
- **Implemented Using Object Containment**
- **Encapsulation**

Composition vs. Aggregation

Feature	Composition	Aggregation
Relationship	Strong "Part-Of" relationship	Weak "Has-A" relationship
Lifespan	The contained object (part) is dependent on the container object. If the container is destroyed, the part is destroyed too.	The contained object (part) can exist independently of the container object.
Ownership	The container owns the contained objects and manages their lifecycle.	The container has a reference to the contained objects but does not own them.
Implementation	Typically uses **direct containment** (i.e., the part is defined as a member variable of the container).	Typically uses **pointers** or **references** to contain objects.
Example	A Car has an Engine, and the Engine is part of the Car. If the Car is destroyed, the Engine is destroyed too.	A Library has Books, but the Books can exist outside the Library. If the Library is destroyed, the Books are not.

```
class Engine {
private:
    std::string engineType;

public:
    // Constructor to initialize the engine type
    Engine(std::string type) : engineType(type) {}

    // Method to get the engine type
    std::string getEngineType() const { return engineType; }
};
```

```
class Car {
private:
    std::string carModel;
    Engine engine; // Composition: Car contains an Engine (part of the Car)

public:
    // Constructor to initialize the car model and the engine type
    Car(std::string model, std::string engineType)
        : carModel(model), engine(engineType) {}

    // Method to display car and engine information
    void display() const {
        std::cout << "Car Model: " << carModel
                  << ", Engine Type: " << engine.getEngineType() << std::endl;
    }
};
```

```
int main() {
    // Create a Car object with an Engine (composition)
    Car myCar("Honda Civic", "V6 Engine");

    // Display car and engine information
    myCar.display();

    return 0;
}
```