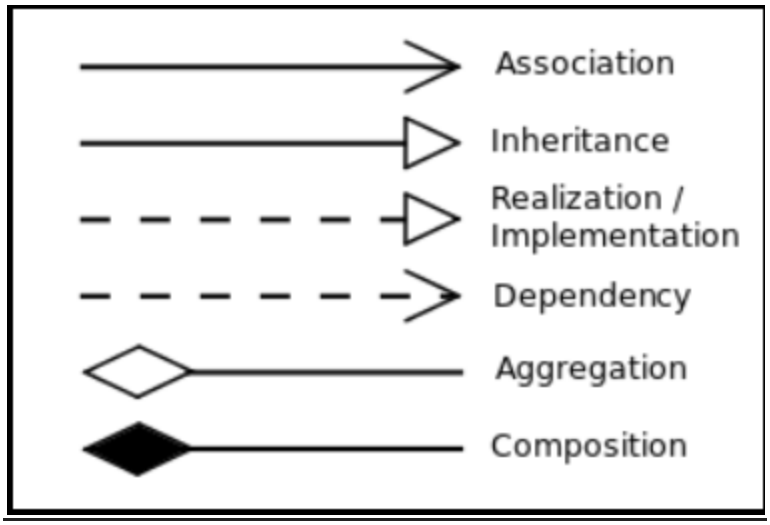
Relationships

A relationship is a general term covering the specific types of logical connections found on class and objects diagrams. UML defines the following relationships:



Instance-level relationships:

- Dependency (always unidirectional)
- Association (Unidirectional or Bi-directional) There are two specialized cases of association:
    - Aggregation
    - Composition

Class-level relationships

- Generalization/Inheritance
- Realization/Implementation

Dependency:

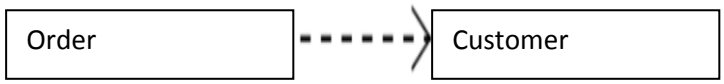
A dependency is a semantic connection between dependent and independent model elements. It exists between two elements if changes to the definition of one element (the server or target) may cause changes to the other (the client or source). ***This relationship is unidirectional.***

- ***Sometime it is called unidirectional association as well***- A slightly less common relationship between two classes. One class is aware of the other and interacts with it.
- Dependency is represented by a dashed arrow starting from the dependent class to its dependency.
- *One class depends on* ***another if the independent class is a parameter variable or local variable of a method of the dependent class****.*
- Sometimes the relationship between two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments.

- ***Multiplicity normally doesn't make sense on a Dependency.***

Example: 1

In below example, we have a Customer class and an Order class. When we need to save a new order, we need to save it corresponding to a customer. In order to do so, our Order class will need a reference to the Customer class and save its data. So in this case, ***our Order class is dependent on the Customer class***. In the future, if any changes are made to the Customer class, it may result in changes to the Order class. See the code below:




```
public class Customer
{
    public Guid CustomerId { get; set; }
    public String CustomerName { get; set; }

    // Other Customer related functions & properties
}

public class Order
{
    public Int32 OrderId { get; set; }
    public Guid OrderCustomerId { get; set; }
    public DateTime OrderDateTime { get; set; }

    // Other Order functions & properties

    public Order(Customer customer)
    {
        // Save order with CustomerId
        this.OrderCustomerId = customer.CustomerId;
    }
}
```

Example: 2

Player is dependent on Dies


```
class Die { public void Roll() { ... } }
class Player
{
    void TakeTurn(Die die)   //Look, I am dependent on Die and it's Roll method to do my work
      {
        die.Roll();
        ......
      }
}
```

Association:

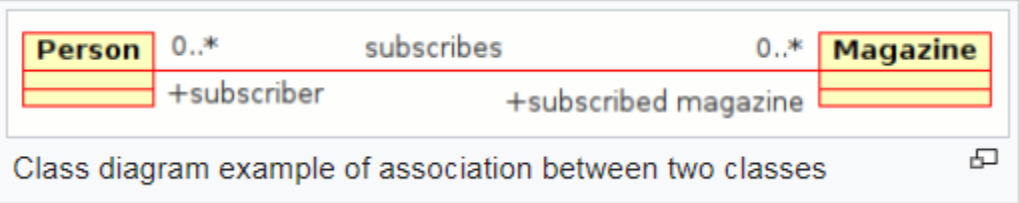
This kind of relation is also referred to as a using relationship, where one class instance uses the other class instance or vice-versa, or both may be using each other. But the main point is, ***the lifetime of the instances of the two classes is independent of each other and there is no ownership between two classes.***

- A binary association (with two ends) is normally represented as a line. An association can link any number of classes. An association with three links is called a ***ternary association.***
- An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.
- No arrow or arrow on both sides that means information can flow into both directions and called bidirectional association.
- Association represents the static relationship shared among the objects of two classes.
- Type of ***has-a*** relationship.

- Multiplicity chart

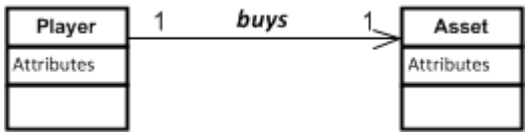
0	No instances (rare)
0..1	No instances, or one instance
1	Exactly one instance
1..1	Exactly one instance
0..*	Zero or more instances
*	Zero or more instances
1..*	One or more instances

Example: 1



Class diagram example of association between two classes

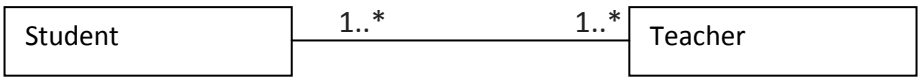
Example: 2




```
class Asset { ... }

class Player {
   Asset asset; /*Set the asset via Constructor or a setter*/
   public Player(Assest purchasedAsset) { ... }
  }
```

Example: 3

Consider the example of a Student-Teacher relationship. Conceptually speaking, each student can be associated with multiple teachers and each teacher can be associated with multiple students. Now to explain this relationship in terms of Object Oriented Programming, see the code below:




```
class Student
{
	string name;
	int rollNum;
	list<Teacher> lstOfTechers;
public:
	void setAssociatedTeachers(list<Tearcher> lstOfTeacher) {
		this->lstOfTeacher = lstOfTeacher;
	}

}

class Teacher
{
	string name;
	int TechId;
	list<Student> lstOfStudent;
}


int main()
{
	Student student = new Student("Rajeev");  //Assume appropriate constructor is
                                                  //available.
	list<Teacher> teachersList;

	Teacher teacher1 = new Teacher("Vinaa");
	Teacher teacher2 = new Teacher("Partha");

	teachersList.add(teacher1);
	teachersList.add(teacher2);

	stu1.setAssociatedTeachers(teachersList);

	//We can reverse the relation ship as well. multiple student can associated
	//with same teacher also.
}
```

Here the student Rajeev associated with multiple teachers like Vinaa & Partha.

- First, both teacher Vinaa & Partha create outside of student class. So their lifetime is independent of the lifetime of the instance of the student (Rajeev).So even if student(Rajeev) is disposed of explicitly, still have teacher class instances teacher1 (Vinaa) & teacher2 (Partha) as alive.

- Secondly, any other student instance, say student2 (Vikash), can also have the same instances of teacher, in other words teacher1 (Vinaa) and teacher2 (Partha) associated with it. So we can also say that no instance of student is a parent of any instance of teacher. So there is no ownership of instances in the association.

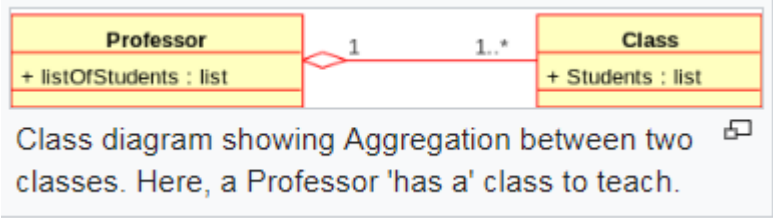
- Similarly, we can have the reverse case, where we have a teacher instance teacher1 (Vinaa) associated with student (Rajeev) and student2 (Vikash), and so on.

Aggregation

Aggregation is a type of association but with an additional point that there is an ownership of the instances, unlike association where there was no ownership of the instances. Aggregation is also referred to as a Weak Association.

- Aggregation is a variant of the ***has A*** association relationship or we can say it is more specific then association.
- An aggregation may not involve more than two classes; it must be a binary association.
- Furthermore, there is hardly a difference between aggregations and associations during implementation, and the diagram may skip aggregation relations altogether.
- ***Aggregation can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. The contents of the container still exist when the container is destroyed.***
- In UML, it is graphically represented as a hollow diamond shape on the containing class with a single line that connects it to the contained class.
- A common perception is that aggregation represents one-to-many / many-to-many / part-whole relationships (i.e. higher multiplicity), which of course can be represented by via association too.

Example: 1



Class diagram showing Aggregation between two classes. Here, a Professor 'has a' class to teach.

Example: 2

To understand it better, let's add another class named Department to our explained example above.

If we talk about the relation between Teacher and Department conceptually, a Department can have multiple Teachers associated with it but each Teacher can belong to only one Department at a time. Now to explain this relationship in terms of Object Oriented Programming, see the code below:

```
class Teacher
{
	string name;
	int TechId;
	list<Student> lstOfStudent;
public:
	Teacher(const string &nm):name(nm){}
};


class Department {

	string  name;
	list<Teacher> teacherWithInDept;
public:

	Department(const string &nm):name(nm){}
	void setAssociatedTeachers(list<Teacher> lstOfTeacher) {
		this->teacherWithInDept = lstOfTeacher;
	}

};
```

```
Main(){

        list<Teacher> teachersList;
        Teacher teacher1("Vinaa");
        Teacher teacher2("Partha");
        teachersList.push_back(teacher1);
        teachersList.push_back(teacher2);
        Department dept1("CS");
        dept1.setAssociatedTeachers(teachersList);
}
```

Here, we have a department instance dept1(CS) and multiple instances of teacher, in other words teacher1(Vinaa) and teacher2(Partha).

- First, the lifetime of Vinaa and Partha are independent of the lifetime of a CS department, because they are instantiated outside the department class. So even if a department CS is disposed still teachers like Vinaa and Partha may continue to exist.

- Secondly, CS department can have multiple teachers associated with it, but the reverse is not true. In other words Vinaa and Partha can belong to only CS only (logically). They cannot belong to any other department like Mechanic. ***So CS department will become the owner of the both teacher.***

Composition:

This is a special kind of aggregation (a type of Association), but with the additional point that the lifetime of dependent class on the owner class.

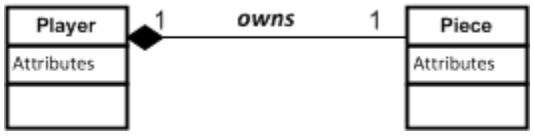
- The UML representation of a composition relationship shows composition as **a *filled diamond shape*** on the containing class end of the lines that connect contained class (es) to the containing class.
- ***When the container is destroyed, the contents are also destroyed, e.g. a university and its departments.***
- Composition is also referred to as a Strong Association or Death relationship.

Example: 1

An `Apartment` is composed of some `Room`s. A `Room` cannot exist without an `Apartment`. When an apartment is deleted, all associated rooms are deleted as well.

```
public class Apartment{
    private Room bedroom;
    public Apartment() {
       bedroom = new Room();
    }
}
```

Example: 2




```
public class Piece { ... }
public class Player
{
        Private pieace;
        Public Player(){
             Piece piece = new Piece();             //Player owns the responsibility
        }                                           //of creating the Piece
...
}
```

Example: 3

To the same code above, let's add another class named University.

So in this case also, conceptually, a university can have multiple departments in it. But each department can belong to a single university only. Now to explain this relationship in terms of Object Oriented Programming, see the code below:

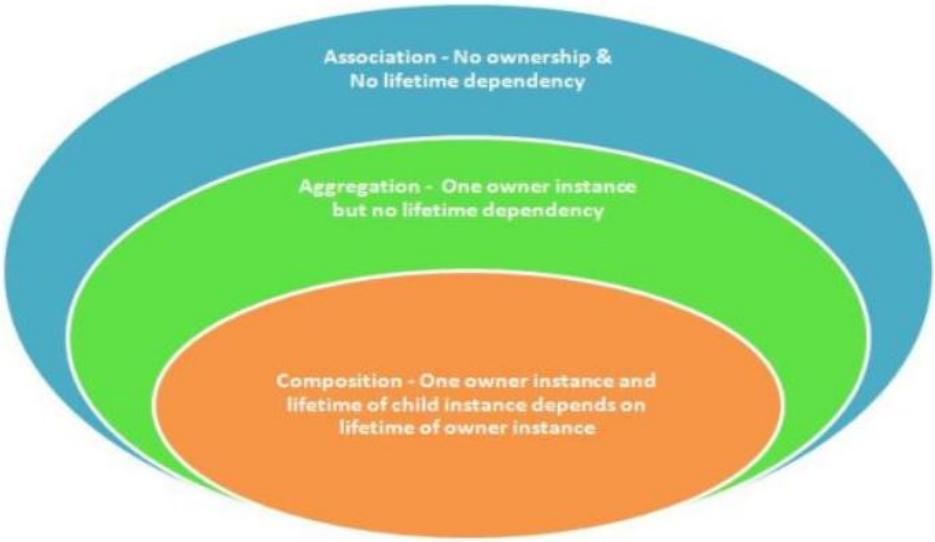
```
class University
{
       string name;
       list<Department> listofDepts;
public:
       University(const string nm):name(nm){}

       void createDepartments() {
              Department dept1("CS");
              Department dept2("Electronics");
              Department dept3("Machanics");
```

```
                listofDepts.push_back(dept1);
                listofDepts.push_back(dept2);
                listofDepts.push_back(dept3);
        }
};

Int main(){

        University uni1("BIT");
        uni1.createDepartments();
}
```

Now observe the below points:

- First, each department like CS, Electronics etc can belong only to a single university instance at a time, BIT. But a university BIT can have multiple departments attached to it. So this makes the university BIT become the owner of the given depts. So this is the combined feature of the aggregation in composition.
- Secondly, here the lifetime of departments which are dependent on the university BIT, because they are being created inside the university class. So when the university BIT is disposed of then all the associated departments are also killed. This is the composition concept.

So if we closely observe the concepts of Association, Aggregation and Composition, we can say that composition is a subset of association and aggregation and aggregation is a subset of association. In other words, Association is a super-set of aggregation and composition can be represented as:



Class-level relationships:

Generalization/Inheritance

It indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*) and the super class is considered a Generalization of the subclass.

For Example: Humans are a subclass of simian, which is a subclass of mammal, and so on. The relationship is most easily understood by the phrase 'an A is a B'

The UML graphical representation of a Generalization is a hollow triangle shape on the super-class end of the line (or tree of lines) that connects it to one or more subtypes.

The generalization relationship is also known as the inheritance or "is a" relationship.

The super class (base class) in the generalization relationship is also known as the "parent". The subtype in the specialization relationship is also known as the "child".

Note that this relationship bears no resemblance to the biological parent–child relationship: the use of these terms is extremely common, but can be misleading.

A is a type of B
For example, "an oak is a type of tree", "an automobile is a type of vehicle"

Generalization can only be shown on class diagrams and on use case diagrams.