UML

**Association**

**Association** is reference based relationship between two classes. Here a class A holds a class level reference to class B. Association can be represented by a line between these classes with an arrow indicating the navigation direction.

Association defines dependency, but a much stronger dependency than that described above with the plain *dependency* relationship. The arrowhead means that there is a one-way relationship. In this example it means that class A1 is associated with class B1. In other words, class A1 uses and contains one instance of class B1, but B1 does not know about or contain any instances of class A1. This example manifests itself as the following Java code:

import B1;

public class A1 {

private B1 b1;

public B1 getB1() {

return b1;

}

}

Association is a relationship between objects. In other words, association defined the multiplicity between objects. You may be aware of one-to-one , one-to-many, many-to-one , many-to-many, all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

Symbol :

|  |  |  |
| --- | --- | --- |
| 0..1 | No instances or one instance | A flight seat can have no or one passenger only |
| 1 | Exactly one instance | An order can have only one customer |
| 0..\* or \* | Zero or more instances | A class can have zero or more students. |
| 1..\* | One or more instances (at least one) | A flight can have one or more passenger |

The unidirectional relationship shows that the source object can invoke methods of the destination class. In Java a possible example can be the instance variable of source class referencing the destination class. Association Example

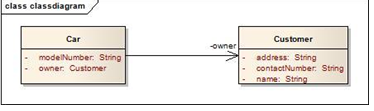
public class Customer {

private String name;

private String address;

private String contactNumber;

}



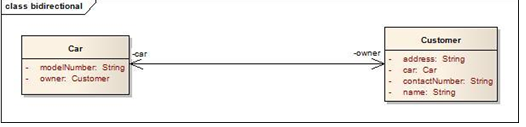
public class Car {

private String modelNumber;

private Customer owner;

}

Let’s look at an example of bidirectional association: Bidirectional association



public class Car {

private String modelNumber;

private Customer owner;

}

public class Customer {

private String name;

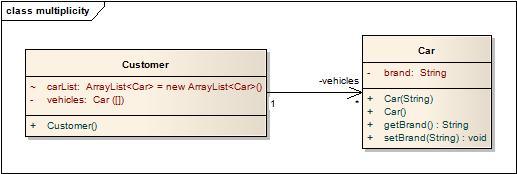
private String address;

private String contactNumber;

private Car car;

}

In the bidirectional association each of the class in this relationship refers to each other by calling each others method. In the above Java example it is depicted as instance variable of Car class in called inside the Customer class and vice versa. In the above example the car and owner refers to the roles and is depicted by the name of instance variable in the code.  
**Multiplicity:** Assume a scenario where a customer has multiple cars. How do we represent this situation in Java and UML?

[](http://i2.wp.com/idiotechie.com/wp-content/uploads/2012/12/multiplicity-in-association.jpg)

Multiplicity in association

The above diagram explains a unidirectional association with a one to may relationship. Both use of ArrayList and Array is for illustration purposes only.

**public class Customer {**

**private Car[] vehicles;**

**ArrayList<Car> carList = new ArrayList<Car>();**

**public Customer(){**

**vehicles = new Car[2];**

**vehicles[0] = new Car("Audi");**

**vehicles[1] = new Car("Mercedes");**

**carList.add(new Car("BMW"));**

**carList.add(new Car("Chevy"));**

**}**

**}**

**public class Car {**

**private String brand;**

**public Car(String brands){**

**this.brand = brands;**

**}**

**public Car() {**

**}**

**public String getBrand() {**

**return brand;**

**}**

**public void setBrand(String brand) {**

**this.brand = brand;**

**}**

**}**

An association is a "has a" relationship. It implies that one class retains a long lasting relationship to another object. Most of the times this implies that a class has an attribute ( a field member) which is the type of the second class, then A has a B. Example will be Class Man uses a Class Pen for writing.

Class Client{

protected String name;

}

An association is **used when** one object wants another object to perform a service for it. Examples are given.

* Employee uses BusService for transportation.
* Client-Server model.
* Computer uses keyboard as input device.

Java code Example is given below.

**public class KeyBoard {**

**public String getCharacters() {**

**return "abcd";**

**}**

**}**

**public class Computer {**

**private KeyBoard keyBoard;**

**public Computer() {**

**keyBoard = new KeyBoard();**

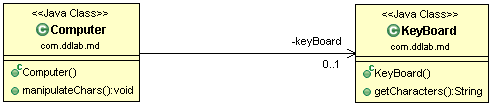
**}**

**public void manipulateChars() {**

**String chars = keyBoard.getCharacters();**

**}**

**}**



**An association is used when one object wants another object to perform a service for it. Examples**

**Computer uses keyboard . For what purpose ? for typing**

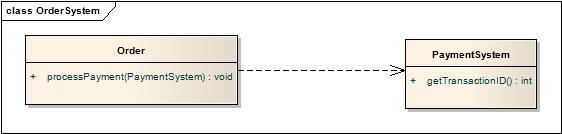
**Employee uses Bus Service. For what purpose ? for transportation**

**Dependency**

Let's start with a dependency. The dependency is a relationship of use, and typically it is not in the diagrams.

Symbol :

Dependency is a relationship that shows that a class is dependent on another class for its existence or implementation. Dependency relationship is shown as a dotted line with an arrow from source class to the dependent class.  
In Java we can consider the dependency relationship if the source class has a reference to the dependent class directly or source class has methods through which the dependent objects are passed as a parameter or refers to the static operation’s of the dependent class or source class has a local variable referring to the dependent class etc.

[](http://i0.wp.com/idiotechie.com/wp-content/uploads/2012/12/Dependency.jpg)

The above diagram satisfies dependency relationship as the source class Order passes the PaymentSystem reference through the processPayment().

public class Order {

public void processPayment(PaymentSystem ps) {

}

}

public class PaymentSystem {

}

The dependency is a relationship of use, and typically it is not in the diagrams.

If A---->B that means that a method in A has a parameter of type B, or that a method of A creates an instance of B to some task and then destroy it. There is just a relation of use, there is not permanent relation between A and B. Of course, there are other kind of dependencies, but these are the most common.

public class MyActionListener implements ActionListener{

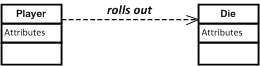
public void actionPerformed(ActionEvent event) {

}

}

Dependency is a loose form of Association. It is "optional" to show dependency in class diagrams. It is represented by a dotted arrow.

**Dependency** is often confused as Association. Dependency is normally created when you receive a reference to a class as part of a particular operation / method. Dependency indicates that you may invoke one of the APIs of the received class reference and any modification to that class may break your class as well. Dependency is represented by a dashed arrow starting from the dependent class to its dependency. Multiplicity normally doesn’t make sense on a Dependency.

[](https://nirajrules.files.wordpress.com/2011/07/dependency.png)

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

die.Roll(); ...

}

}

Dependency is a loose form of Association. It is "optional" to show dependency in class diagrams. The best example will be a **Teacher teaches a subject of Physics**. The java code is given below.

public class Subject {

private String subjectName = "Physics";

public String getSubjectName() {

return subjectName;

}

}

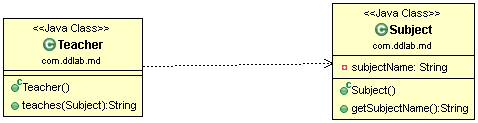
public class Teacher {

public String teaches( Subject subject ) {

return subject.getSubjectName();

}

}



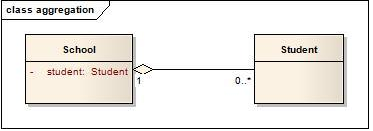
A **dependency** relates to collaboration or delegation, where an object requests services from another object and is therefor dependent on that object. As the client of the service, you want the service interface to remain constant, even if future services are offered.

**Aggregation – ( HAS-A )**

Aggregation is a special case of association. A directional association between objects . When an object “**has-a**” another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called “**Has-a**” relationship.

**Symbol** : http://javapapers.com/wp-content/uploads/2010/06/aggregation.jpg

Example will be a Class Man has a Class Car ( Car is still there when Man dies ) . This shows “has a” relationship. It is a form of association relationship. This relationship highlights that a whole is made of its parts. So if a whole is destroyed the part still remains. In UML this is represented through a hollow diamond with the diamond symbol pointing towards the whole.  
In case of Java the aggregation follows the same structure as association. It is represented through the instance variables of a class.



In this case a student is a part of the School. However during design it is preferred to use association instead of aggregation as it is not a recommended option. Complete java code is given below.

public class Student {

private String name;

private String subject;

public String getSubject() {

return subject;

}

public void setSubject(String subject) {

this.subject = subject;

}

public Student( String name ) {

this.name = name;

}

public String getName() {

return name;

}

}

import java.util.List;

public class School {

private List<Student> studentsList;

public School( List<Student> studentsList ) {

this.studentsList = studentsList;

}

public void provides(String subject) {

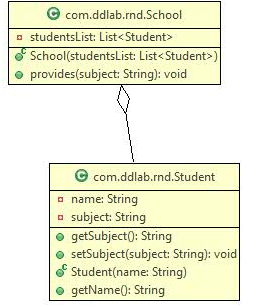
for(Student student : studentsList ) {

student.setSubject(subject);

}

}

}



**Aggregation** is same as association and is often seen as redundant relationship. 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 (hence the redundancy). As aggregation doesn’t convey anything more effective about a software design than an association, there is no separate UML representation for it (though some developers use a hollow diamond to indicate aggregation). You can give aggregation a miss unless you use it to convey something special.

[*Aggregation*](https://en.wikipedia.org/wiki/Aggregation_(object-oriented_programming)) is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As shown in the image, a Professor 'has a' class to teach. As a type of association, an aggregation can be named and have the same adornments that an association can. However, 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 are not automatically destroyed when the container is.

In [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language), it is graphically represented as a *hollow* [diamond shape](https://en.wikipedia.org/wiki/Rhombus) on the containing class with a single line that connects it to the contained class. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

**Composition – ( HAS-A Restricted , Interdependent )**

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition. Ex:a Class Man owns a Class Heart ( When Man die, Heart die )

Symbol : http://javapapers.com/wp-content/uploads/2010/06/composition.jpg

Example : A class contains students. A student cannot exist without a class. There exists composition between class and students.

**Differences**

**Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

**Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

The above two are forms of **containment** (hence the parent-child relationships).

**Dependency** is a weaker form of relationship and in code terms indicates that a class uses another by parameter or return type.

Dependency is a form of **association**.

Another interpretation

aggregation and composition are almost completely identical except that **composition** is used when the life of the child is completely controlled by the parent.

**Aggregation**

Car->Tires

The tires can be taken off of the car object and installed on a different one. Also, if the car gets totaled, the tires do not necessarily have to be destroyed.

**Composition**

Body->Blood Cell

When the Body object is destroyed the BloodCells get destroyed with it.

**Dependency**

A relationship between two objects where changing one may affect the other.

**Aggregation** - separable part to whole. The part has a identity of its own, separate from what it is part of. You could pick that part and move it to another object. (real world examples: wheel -> car, bloodcell -> body)

**Composition** - non-separable part of the whole. You cannot move the part to another object. more like a property. (real world examples: curve -> road, personality -> person, max\_speed -> car, property of object -> object )

Note that a relation that is an aggregate in one design can be a composition in another. Its all about how the relation is to be used in that specific design.

A **dependency** relates to collaboration or delegation, where an object requests services from another object and is therefore dependent on that object. As the client of the service, you want the service interface to remain constant, even if future services are offered.

Composition is more restrictive. When there is a c composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies , which object contains the other object.

Example: A library contains students and books. Relationship between library and student is aggregation. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore it is a composition. For easy understanding I am picking this example.

Composite aggregation is depicted as a binary association decorated with a **filled black diamond** at the aggregate (whole) end.

Composite aggregation is depicted as binary associations with filled black diamond

Folder could contain many files, while each File has exactly one Folder parent. If Folder is deleted, all contained Files are deleted as well. In cases where in addition to the part-of relationship between ClassA and ClassB - there’s a strong life cycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result, we should be more specific and use the composition link instead of the aggregation link or the association link. The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts), meaning that the container object and its parts constitute a parent-child/s relationship. Java code for Composition is given below.

public class Heart {

private String condition;

public String getCondition() {

return condition;

}

}

public class Head {

private String condition;

public String getCondition() {

return condition;

}

}

[](http://lh5.ggpht.com/_aUOgqE3fGXc/Sh35YNKDw9I/AAAAAAAAAao/E4v4uDJcD5w/s1600-h/image%5b5%5d.png)

public class Person {

private Heart heart;

private Head head;

private Leg[] legs;

public Person() {

heart = new Heart();

head = new Head();

legs = new Leg[2];

}

}

public class Leg {

private String leftLegName;

private String rightLegName;

public String getLeftLegName() {

return leftLegName;

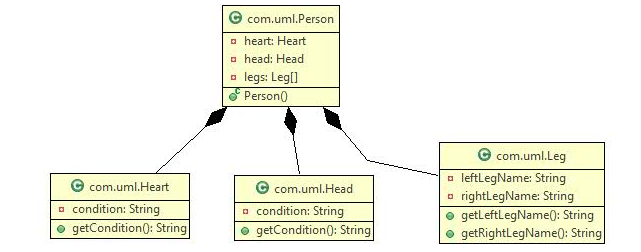
}

public String getRightLegName() {

return rightLegName;

}

}



**Generalization – ( IS-A ) – Inheritance**

Generalization uses a “**IS-A**” relationship from a specialization to the generalization class. Common structure and behavior are used from the specialization to the generalized class. At a very broader level you can understand this as inheritance. Generalization is also called a “**IS-A**” relationship.

Example: Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here is the relationship between student and person, similarly faculty and person is generalization.

Symbol : http://javapapers.com/wp-content/uploads/2010/06/generalization.jpg

Java code for Generalization is given below.

**public** **class** Bird {

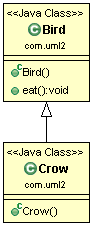
**public** **void** eat() {

}

}

**public** **class** Crow **extends** Bird {

}



Another example of Generalization is types of account in a bank. The code is given below. The diagram is given below.

**public** **class** Account {

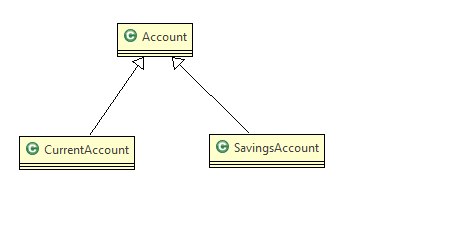
}

**public** **class** CurrentAccount **extends** Account {

}

**public** **class** SavingsAccount **extends** Account {

}



**Realization ( implements )**

This is related to the relationship between the class and the interface. The realization is equivalent to the “***implements***” keyword in Java. Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

Symbol http://javapapers.com/wp-content/uploads/2010/06/realization.jpg

Java code for Realization

**public** **class** **AccountValidatorImpl** **implements** Validator {

@Override

**public** **boolean** isValid() {

**return** **false**;

}

}

**public** **interface** **Validator** {

**public** **boolean** isValid();

}

**public** **class** LoanValidatorImpl **implements** Validator {

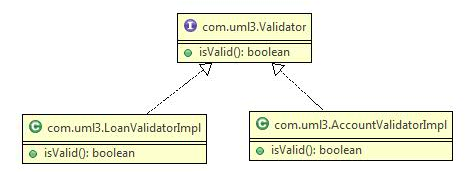
@Override

**public** **boolean** isValid() {

**return** **false**;

}

}



Tools Used for UML Designing

1. Visual Paradigm
2. Enterprise Arhitect
3. Astah
4. Edraw Max
5. Gliffy
6. Lucidchart
7. [Modelio](https://en.wikipedia.org/wiki/Modelio)
8. [yEd](https://en.wikipedia.org/wiki/YEd)
9. Dia

Eclipse Plugins

1. Object Aid
2. Amateras UML
3. UMlet
4. Obeo UML Designer