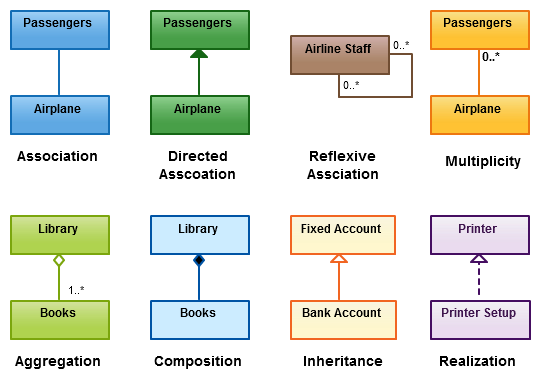
# Class Diagram Relationships in UML with Examples

Many people consider class diagrams a bit more complicated to build compared with ER diagrams. While this might be true, this article helps clip some of the complexities of class diagrams in such a way that even non-programmers and less tech-savvy individuals will come to appreciate the usefulness of this modeling approach. In particular, this article explains how to correctly determine and implement the different class diagram relationships that are applicable in object-oriented modeling.

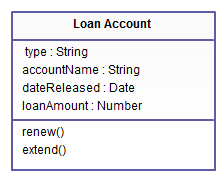
**[](http://static3.creately.com/blog/wp-content/uploads/2012/03/Class-Diagram-Relationships.png)**

*Relationships in UML class diagrams*

### Class Diagrams Explained

Class diagrams are visual representations of the static structure and composition of a particular system using the conventions set by the Unified Modeling Language (UML). Out of all the [**UML diagram types**](http://creately.com/blog/diagrams/uml-diagram-types-examples/) it is one of the most used ones. System designers use class diagrams as a way of simplifying how objects in a system interact with each other. Using class diagrams, it is easier to describe all the classes, packages, and interfaces that constitute a system and how these components are interrelated. For example, a simple class diagram may be used to show how an organization such as a convenient store chain is set up. On the other hand, precisely detailed class diagrams can readily be used as the primary reference for translating the designed system into a programming code.

The following figure is an example of a simple class diagram:

**[](http://static3.creately.com/blog/wp-content/uploads/2012/03/Class-Diagram.jpeg)**

*Simple class diagram with attributes and operations*

In the example, a class called “loan account” is depicted. Classes in class diagrams are represented by boxes that are partitioned into three:

1. The top partition contains the name of the class.
2. The middle part contains the class’s attributes.
3. The bottom partition shows the possible operations that are associated with the class.

Those should be pretty easy to see in the example: the class being described is a loan account, some of whose attributes include the type of loan, the name of the borrower/loaner, the specific date the loan was released and the loan amount. As in the real world, various transactions or operations may be implemented on existing loans such as renew and extend.  The example shows how class diagrams can encapsulate all the relevant data in a particular scenario in a very systematic and clear way.

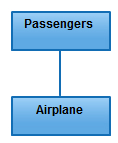
In object-oriented modeling, class diagrams are considered the key building blocks that enable information architects, designers, and developers to show a given system’s classes, their attributes, the functions or operations that are associated with them, and the relationships among the different classes that make up a system.

**Relationships in Class Diagrams**

Classes are interrelated to each other in specific ways. In particular, relationships in class diagrams include different types of logical connections. The following are such types of logical connections that are possible in UML:

* [**Association**](http://creately.com/blog/diagrams/class-diagram-relationships/#Association)
* [**Directed Association**](http://creately.com/blog/diagrams/class-diagram-relationships/#Directed)
* [**Reflexive Association**](http://creately.com/blog/diagrams/class-diagram-relationships/#Reflexive)
* [**Multiplicity**](http://creately.com/blog/diagrams/class-diagram-relationships/#Multiplicity)
* [**Aggregation**](http://creately.com/blog/diagrams/class-diagram-relationships/#Aggregation)
* [**Composition**](http://creately.com/blog/diagrams/class-diagram-relationships/#Composition)
* [**Inheritance/Generalization**](http://creately.com/blog/diagrams/class-diagram-relationships/#Inheritance)
* [**Realization**](http://creately.com/blog/diagrams/class-diagram-relationships/#Realization)

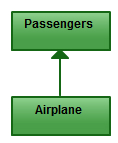
### Association

**[](http://static3.creately.com/blog/wp-content/uploads/2012/03/Association-Relationship.jpeg)**

**Association**

is a broad term that encompasses just about any logical connection or relationship between classes. For example, passenger and airline may be linked as above:

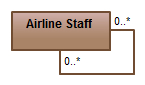
### *Directed Association*

**[](http://static2.creately.com/blog/wp-content/uploads/2012/03/Directed-Association-Relationship.jpeg)**

**Directed Association**

refers to a directional relationship represented by a line with an arrowhead. The arrowhead depicts a container-contained directional flow.

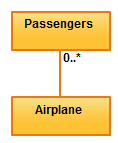
### *Reflexive Association*

**[](http://static2.creately.com/blog/wp-content/uploads/2012/03/Reflexive-Association-Relationship.jpeg)**

**Reflexive Association**

occurs when a class may have multiple functions or responsibilities. For example, a staff working in an airport may be a pilot, aviation engineer, a ticket dispatcher, a guard, or a maintenance crew member. If the maintenance crew member is managed by the aviation engineer there could be a managed by relationship in two instances of the same class.

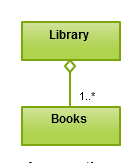
### *Multiplicity*

**[](http://static3.creately.com/blog/wp-content/uploads/2012/03/Multiplicity-Relationship.jpeg)**

**Multiplicity**

is the active logical association when the cardinality of a class in relation to another is being depicted. For example, one fleet may include multiple airplanes, while one commercial airplane may contain zero to many passengers. The notation 0..\* in the diagram means “zero to many”.

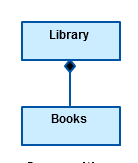
### Aggregation

**[](http://static1.creately.com/blog/wp-content/uploads/2012/03/Aggregation-Relationship.png)**

**Aggregation**

refers to the formation of a particular class as a result of one class being aggregated or built as a collection. For example, the class “library” is made up of one or more books, among other materials. In aggregation, the contained classes are not strongly dependent on the life cycle of the container. In the same example, books will remain so even when the library is dissolved. To render aggregation in a diagram, draw a line from the parent class to the child class with a diamond shape near the parent class.

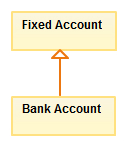
### Composition

**[](http://static1.creately.com/blog/wp-content/uploads/2012/03/Composition-Relationship-UML.png)**

**Composition**

is very similar to the aggregation relationship, with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class. That is, the contained class will be obliterated when the container class is destroyed. For example, a shoulder bag’s side pocket will also cease to exist once the shoulder bag is destroyed. To depict a composition relationship in a UML diagram, use a directional line connecting the two classes, with a filled diamond shape adjacent to the container class and the directional arrow to the contained class.

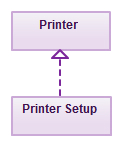
### Inheritance / Generalization

**[](http://static2.creately.com/blog/wp-content/uploads/2012/03/Inheritance-Relationship.jpeg)**

**Inheritance**

refers to a type of relationship wherein one associated class is a child of another by virtue of assuming the same functionalities of the parent class. In other words, the child class is a specific type of the parent class. To depict inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead.

### Realization

**[](http://static2.creately.com/blog/wp-content/uploads/2012/03/Realization-Relationship.jpeg)**

**Realization**

denotes the implementation of the functionality defined in one class by another class. To show the relationship in UML, a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality to the class that implements the function. In the example, the printing preferences that are set using the printer setup interface are being implemented by the printer.

### Conclusion – Class diagram relationships are easy to understand

If you are a programmer or systems designer, you’ll be building or analyzing class diagrams quite often since they are, after all, the building blocks of object-oriented modeling. As demonstrated by this article, class diagram relationships are fairly easy to understand. As a rule of thumb, keeping class diagrams as simple as possible allows them to be more easily understood and appreciated by different types of audiences. For this purpose, remember to label your classes and relationships as descriptive as possible. Lastly, class diagrams also evolve as the real world systems they represent change. This implies that you don’t need to put in much detail in your first draft. All the classes, interfaces and relationships that are integral to the system or application you are designing will eventually emerge as the development process moves forward.

To make your job a lot easier, you can check out the online diagramming application offered on the **[Creately](http://creately.com/diagram-type/class-diagram)** site. Besides being easy to use, the platform provides a comprehensive range of UML templates among other diagramming services. In addition, the platform supports collaboration and may be integrated into an existing company wiki or Intranet to keep diagrams well documented and updated. Having such a tool on your side will greatly improve your company’s development initiatives and will help your team meet its targets.

Now that you understand class diagram relationships it’s time to draw some, get started with this [**easy to use class diagram templates**](http://creately.com/diagram-community/examples/t/class-diagram).

**References:**

1. [***UML basics: The class diagram An introduction to structure diagrams in UML 2 by Donald Bell***](http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/)

2. [**Class diagramas published on the Wikipedia website**](http://en.wikipedia.org/wiki/Class_diagram)

3. [**The UML Class Diagram Part 1 as published in the website developer.com**](http://www.developer.com/design/article.php/2206791/The-UML-Class-Diagram-Part-1.htm)

4. [**The Class Diagram fromVisual Case Tool – UML Tutorialas published on Visual Case website**](http://www.visualcase.com/tutorials/class-diagram.htm)

5.  [**Associations as published on the Sybase website**](http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc38086.1530/doc/html/rad1232632566113.html)

# Association, Aggregation, Composition, Abstraction, Generalization, Realization, Dependency

These terms signify the relationships between classes. These are the building blocks of object oriented programming and very basic stuff. But still for some, these terms look like Latin and Greek. Just wanted to refresh these terms and explain in simpler terms.

## Association

Association is a relationship between two objects. In other words, association defines 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.

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

**Example:**A Student and a Faculty are having an association.

## Aggregation

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 a “Has-a” relationship.

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

## Composition

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.

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.

### Difference between aggregation and composition

Composition is more restrictive. When there is a 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. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition. For easy understanding I am picking this example. Don’t go deeper into example and justify relationships!

## *Abstraction*

Abstraction is specifying the framework and hiding the implementation level information. Concreteness will be built on top of the abstraction. It gives you a blueprint to follow to while implementing the details. Abstraction reduces the complexity by hiding low level details.

***Example:***A wire frame model of a car.

## *Generalization*

Generalization uses a “is-a” relationship from a specialization to the generalization class. Common structure and behaviour are used from the specializtion to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a “Is-a” relationship.

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

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

## *Realization*

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.

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

***Example:*** A particular model of a car ‘GTB Fiorano’ that implements the blueprint of a car realizes the abstraction.

## *Dependency*

Change in structure or behaviour of a class affects the other related class, then there is a dependency between those two classes. It need not be the same vice-versa. When one class contains the other class it this happens.

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

***Example:***Relationship between shape and circle is dependency.

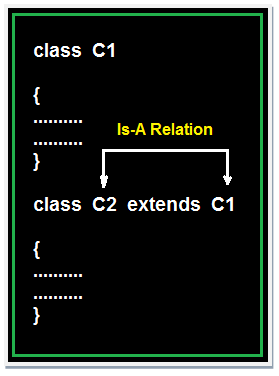
Type of relationship always makes to understand how to reuse the feature from one class to another class. In java programming we have three types of relationship they are.

* Is-A Relationship
* Has-A Relationship
* Uses-A Relationship

### *Is-A relationship*

In Is-A relationship one class is obtaining the features of another class by using inheritance concept with extends keywords.

In a IS-A relationship there exists logical memory space.



### Example of Is-A Relation

## Example

**class** Faculty

{

**float** salary=30000;

}

**class** Science **extends** Faculty

{

**float** bonous=2000;

**public** **static** **void** main(String args[])

{

Science obj=**new** Science();

System.**out**.println("Salary is:"+obj.salary);

System.**out**.println("Bonous is:"+obj.bonous);

}

}

## Output

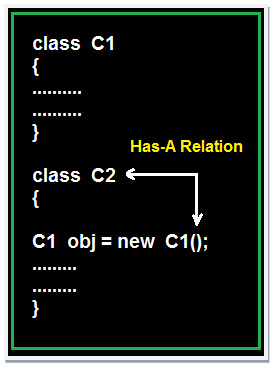
Salary is: 30000.0

Bonous is: 2000.0

### *Has-A relationship*

In Has-A relationship an object of one class is created as data member in another class the relationship between these two classes is Has-A.

In Has-A relationship there existed physical memory space and it is also known as part of or kind of relationship.



### Example of Has-A Relation

## Example

**class** Employee

{

**float** salary=30000;

}

**class** Developer **extends** Employee

{

**float** bonous=2000;

**public** **static** **void** main(String args[])

{

Employee obj=**new** Employee();

System.**out**.println("Salary is:"+obj.salary);

}

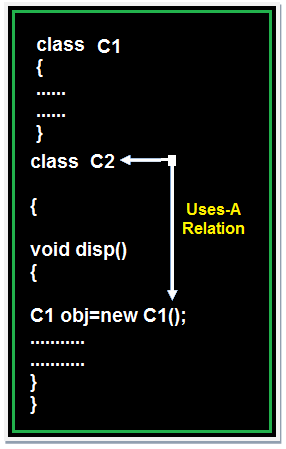
}

## Output

Salary is: 30000.0

### *Uses-A relationship*

A method of one class is using an object of another class the relationship between these two classes is known as Uses-A relationship.



As long as the method is execution the object space (o1) exists and once the method execution is completed automatically object memory space will be destroyed.

### Example of Uses-A Relation

## Example

**class** Employee {

**float** salary=30000;

}

**class** Salary **extends** Employee {

**void** disp() {

**float** bonous=1000;

Employee obj=**new** Employee();

**float** Total=obj.salary+bonous;

System.**out**.println("Total Salary is:"+Total);

}

}

**class** Developer {

**public** **static** **void** main(String args[]) {

Salary s=**new** Salary();

s.disp();

}

}

## Output

Total Salary is: 31000.0

**Note 1:** The default relationship in java is Is-A because for each and every class in java there exist an implicit predefined super class is java.lang.Object.

**Note 2:** The universal example for Has-A relationship is System.out (in System.out statement, out is an object of printStream class created as static data member in another system class and printStream class is known as Has-A relationship).

**Note 3:** Every execution logic method (main() ) of execution logic is making use of an object of business logic class and business logic class is known as Uses-A relationship.

***Description***

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the vimplementation of inheritance (IS-A relationship), or object composition (HAS-A relationship). Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition.

***IS-A Relationship:***

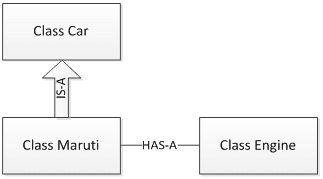
In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

***HAS-A Relationship:***

Composition(HAS-A) simply mean the use of instance variables that are references to other objects. For example Maruti has Engine, or House has Bathroom.

Let’s understand these concepts with an example of Car class.



package relationships;

class Car {

// Methods implementation and class/Instance members

private String color;

private int maxSpeed;

public void carInfo(){

System.out.println("Car Color= "+color + " Max Speed= " + maxSpeed);

}

public void setColor(String color) {

this.color = color;

}

public void setMaxSpeed(int maxSpeed) {

this.maxSpeed = maxSpeed;

}

}

As shown above, Car class has a couple of instance variable and few methods. Maruti is a specific type of Car which extends Car class means Maruti IS-A Car.

class Maruti extends Car{

//Maruti extends Car and thus inherits all methods from Car (except final and static)

//Maruti can also define all its specific functionality

public void MarutiStartDemo(){

Engine MarutiEngine = new Engine();

MarutiEngine.start();

}

}

Maruti class uses Engine object’s start() method via composition. We can say that Maruti class HAS-A Engine.

package relationships;

public class Engine {

public void start(){

System.out.println("Engine Started:");

}

public void stop(){

System.out.println("Engine Stopped:");

}

}

RelationsDemo class is making object of Maruti class and initialized it. Though Maruti class does not have setColor(), setMaxSpeed() and carInfo() methods still we can use it due to IS-A relationship of Maruti class with Car class.

package relationships;

public class RelationsDemo {

public static void main(String[] args) {

Maruti myMaruti = new Maruti();

myMaruti.setColor("RED");

myMaruti.setMaxSpeed(180);

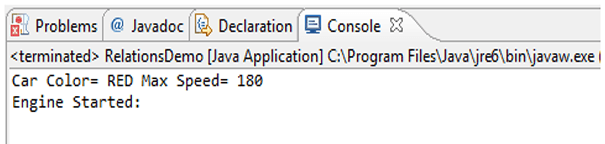
myMaruti.carInfo();

myMaruti.MarutiStartDemo();

}

}

If we run RelationsDemo class we can see output like below.



***Comparing Composition and Inheritance***

* It is easier to change the class implementing composition than inheritance. The change of a superclass impacts the inheritance hierarchy to subclasses.
* You can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
* Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding)
* It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition) because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.
* With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remains inside the same class.
  + **Don't use inheritance just to get code reuse** If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
  + **Don't use inheritance just to get at polymorphism** If all you really want is a polymorphism, but there is no natural is-a relationship, use composition with interfaces.

**Summary**

* IS-A relationship based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance.
* Has-a relationship is composition relationship which is a productive way of code reuse.