What is Class in UML Diagram?

A **Class in UML** diagram is a blueprint used to create an object or set of objects. The Class defines what an object can do. It is a template to create various objects and implement their behavior in the system. A Class in UML is represented by a rectangle that includes rows with class names, attributes, and operations.

What is Class Diagram?

A **Class Diagram** in Software engineering is a static structure that gives an overview of a software system by displaying classes, attributes, operations, and their relationships between each other. This Diagram includes the class name, attributes, and operation in separate designated compartments. Class Diagram helps construct the code for the software application development.
Class Diagram defines the types of objects in the system and the different types of relationships that exist among them. It gives a high-level view of an application. This modeling method can run with almost all Object-Oriented Methods. A class can refer to another class. A class can have its objects or may inherit from other classes.

In this UML Class Diagram Lab, you will learn:

- What is Class?
- What is Class Diagram?
- Benefits of Class Diagram
- Essential elements of A UML class diagram
- Class Name
- Attributes:
- Relationships
- Aggregation vs. Composition
- Abstract Classes
- Example of UML Class Diagram:
- Class Diagram in Software Development Lifecycle
- Best practices of Designing of the Class Diagram

Benefits of Class Diagram

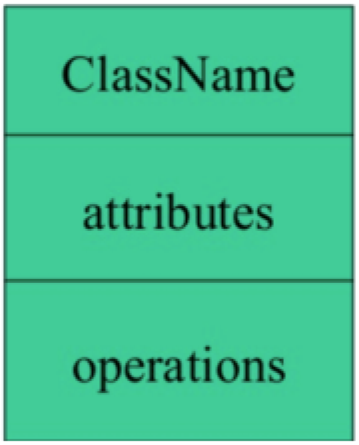
- Class Diagram Illustrates data models for even very complex information systems
- It provides an overview of how the application is structured before studying the actual code. This can easily reduce the maintenance time
- It helps for better understanding of general schematics of an application.
- Allows drawing detailed charts which highlights code required to be programmed
- Helpful for developers and other stakeholders.

Essential elements of A UML class diagram

Essential elements of UML class diagram are:

1. Class Name
2. Attributes
3. Operations

Class Name



The name of the class is only needed in the graphical representation of the class. It appears in the topmost compartment. A class is the blueprint of an object which can share the same relationships, attributes, operations, & semantics. The class is rendered as a rectangle, including its name, attributes, and operations in sperate compartments.

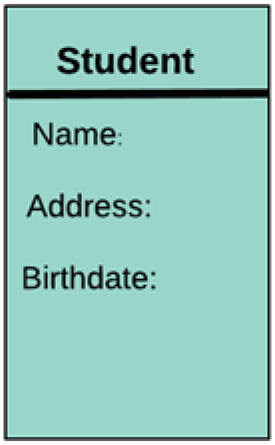
Following rules must be taken care of while representing a class:

1. A class name should always start with a capital letter.

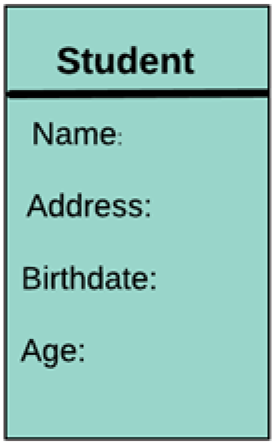
2. A class name should always be in the center of the first compartment.
3. A class name should always be written in **bold** format.
4. UML abstract class name should be written in italics format.

Attributes:

An attribute is named property of a class which describes the object being modeled. In the class diagram, this component is placed just below the name-compartment.



A derived attribute is computed from other attributes. For example, an age of the student can be easily computed from his/her birth date.



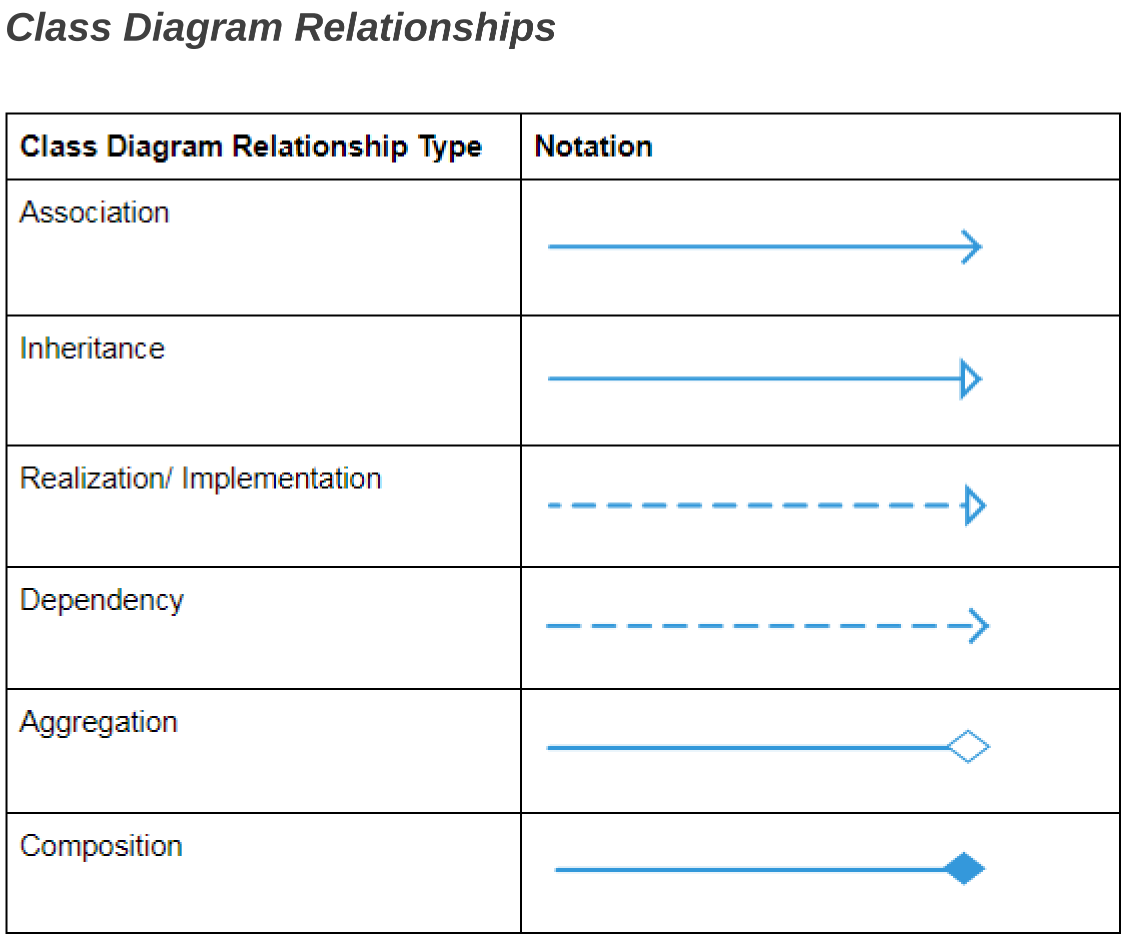
Attributes characteristics

- The attributes are generally written along with the visibility factor.
- Public, private, protected and package are the four visibilities which are denoted by +, -, #, or ~ signs respectively.
- Visibility describes the accessibility of an attribute of a class.
- Attributes must have a meaningful name that describes the use of it in a class.

Relationships

There are mainly three kinds of relationships in UML:

1. Dependencies
2. Generalizations
3. Associations



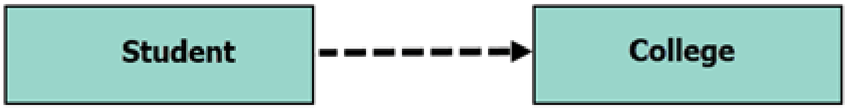
Class Diagram Relationships

Class Diagram Relationship Type	Notation
Association	——→
Inheritance	——▷
Realization/ Implementation	- - - -▷
Dependency	- - - -→
Aggregation	——◇
Composition	——◆

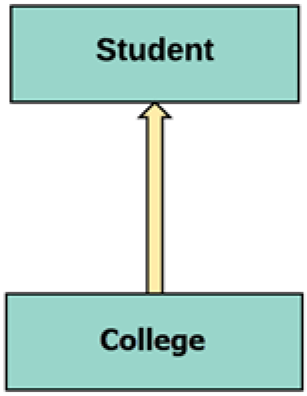
Dependency

A dependency means the relation between two or more classes in which a change in one may force changes in the other. However, it will always create a weaker relationship. Dependency indicates that one class depends on another.

In the following UML class diagram examples, Student has a dependency on College



Generalization:



A generalization helps to connect a subclass to its superclass. A sub-class is inherited from its superclass. Generalization relationship can’t be used to model interface implementation. Class diagram allows inheriting from multiple superclasses.

In this example, the class Student is generalized from Person Class.

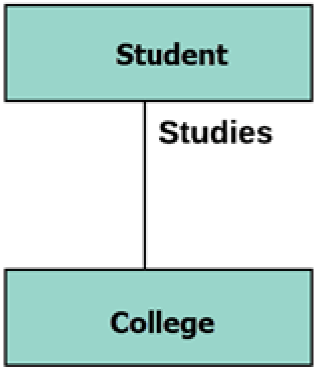
Association:

This kind of relationship represents static relationships between classes A and B. For example; an employee works for an organization.

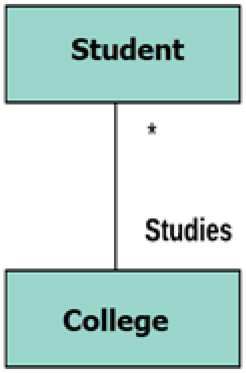
Here are some rules for Association:

- Association is mostly verb or a verb phrase or noun or noun phrase.
- It should be named to indicate the role played by the class attached at the end of the association path.
- Mandatory for reflexive associations

In this example, the relationship between student and college is shown which is studies.



Multiplicity

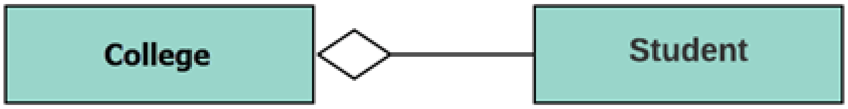


A multiplicity is a factor associated with an attribute. It specifies how many instances of attributes are created when a class is initialized. If a multiplicity is not specified, by default one is considered as a default multiplicity.

Let’s say that that there are 100 students in one college. The college can have multiple students.

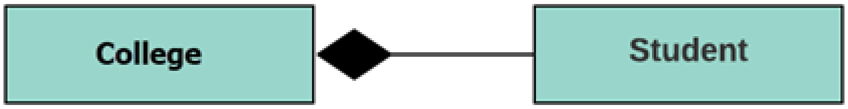
Aggregation

Aggregation is a special type of association that models a whole- part relationship between aggregate and its parts.



For example, the class college is made up of one or more student. In aggregation, the contained classes are never totally dependent on the lifecycle of the container. Here, the college class will remain even if the student is not available.

Composition:



The composition is a special type of aggregation which denotes strong ownership between two classes when one class is a part of another class.

For example, if college is composed of classes student. The college could contain many students, while each student belongs to only one college. So, if college is not functioning all the students also removed.

Aggregation vs. Composition

Aggregation	Composition
Aggregation indicates a relationship where the child can exist separately from their parent class. Example: Automobile (Parent) and Car (Child). So, If you delete the Automobile, the child Car still exist.	Composition display relationship where the child will never exist independent of the parent. Example: House (parent) and Room (child). Rooms will never separate into a House.

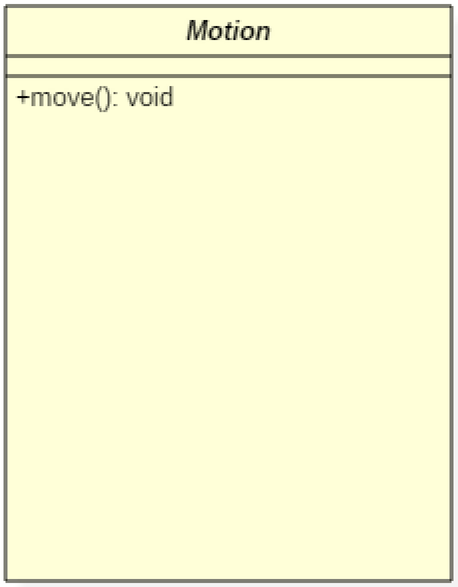
Abstract Classes

It is a class with an operation prototype, but not the implementation. It is also possible to have an abstract class with no operations declared inside of it. An abstract is useful for identifying the functionalities across the classes. Let us consider an example of an abstract class. Suppose we have an abstract class called as a motion with a method or an operation declared inside of it. The method declared inside the abstract class is called a **move ()**.

This abstract class method can be used by any object such as a car, an animal, robot, etc. for changing the current position. It is efficient to use this abstract class method with an object because no implementation is provided for the given function. We can use it in any way for multiple objects.

In UML, the abstract class has the same notation as that of the class. The only difference between a class and an abstract class is that the class name is strictly written in an italic font.

An abstract class cannot be initialized or instantiated.



Abstract Class Notation

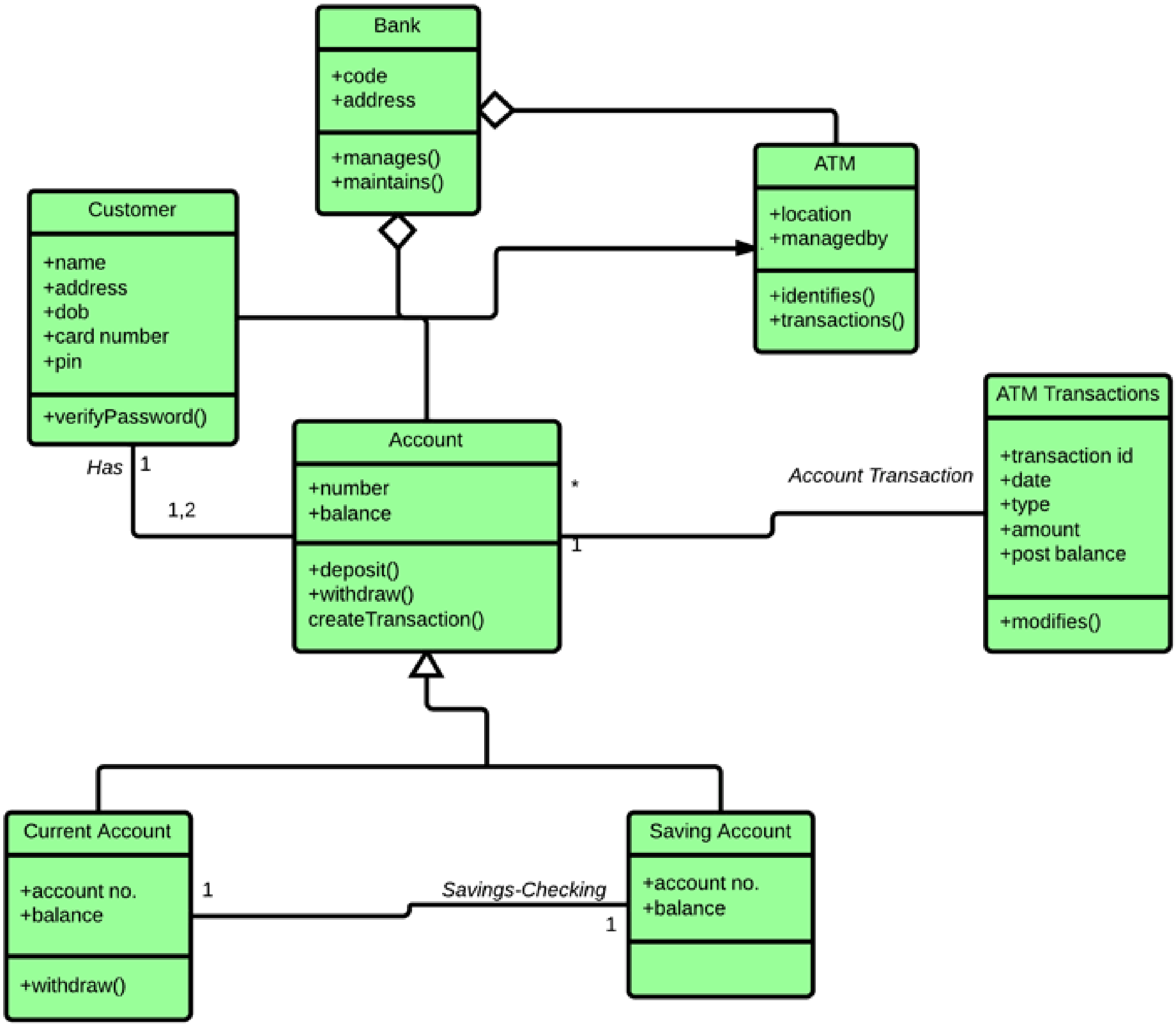
In the above abstract class notation, there is the only a single abstract method which can be used by multiple objects of classes.

Example of UML Class Diagram

Creating a class diagram is a straightforward process. It does not involve many technicalities. Here, is an example:

ATMs system is very simple as customers need to press some buttons to receive cash. However, there are multiple security layers that any ATM system needs to pass. This helps to prevent fraud and provide cash or need details to banking customers.

Below given is a UML Class Diagram example:



UML Class Diagram Example

Class Diagram in Software Development Lifecycle

Class diagrams can be used in various software development phases. It helps in modeling class diagrams in three different perspectives.

1. Conceptual perspective: Conceptual diagrams are describing things in the real world. You should draw a diagram that represents the concepts in the domain under study. These concepts related to class and it is always language-independent.

2. Specification perspective: Specification perspective describes software abstractions or components with specifications and interfaces. However, it does not give any commitment to specific implementation.

3. Implementation perspective: This type of class diagrams is used for implementations in a specific language or application. Implementation perspective, use for software implementation.

Best practices of Designing of the Class Diagram

Class diagrams are the most important UML diagrams used for software application development. There are many properties which should be considered while drawing a Class Diagram. They represent various aspects of a software application.

Here, are some points which should be kept in mind while drawing a class diagram:

- The name given to the class diagram must be meaningful. Moreover, It should describe the real aspect of the system.
- The relationship between each element needs to be identified in advance.
- The responsibility for every class needs to be identified.
- For every class, minimum number of properties should be specified. Therefore, unwanted properties can easily make the diagram complicated.
- User notes should be included whenever you need to define some aspect of the diagram. At the end of the drawing, it must be understandable for the software development team.

- Lastly, before creating the final version, the diagram needs to be drawn on plain paper. Moreover, It should be reworked until it is ready for final submission.

Conclusion

- UML is the standard language for specifying, designing, and visualizing the artifacts of software systems
- A class is a blueprint for an object
- A class diagram describes the types of objects in the system and the different kinds of relationships which exist among them
- It allows analysis and design of the static view of a software application
- Class diagrams are most important UML diagrams used for software application development
- Essential elements of UML class diagram are 1) Class 2) Attributes 3) Relationships
- Class Diagram provides an overview of how the application is structured before studying the actual code. It certainly reduces the maintenance time
- The class diagram is useful to map object-oriented programming languages like Java, C++, Ruby, Python, etc.

Relationships in UML diagram are used to represent a connection between various things. A relationship is a connection amongst things such as structural, behavioral, or grouping things in the unified modeling language.

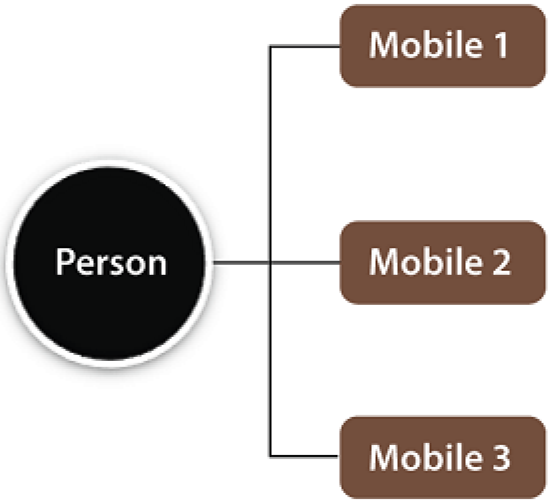
Following are the different types of standard relationships in UML,

- Association
- Dependency
- Generalization
- Realization

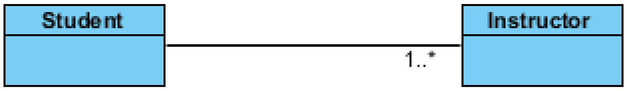
Association in Java

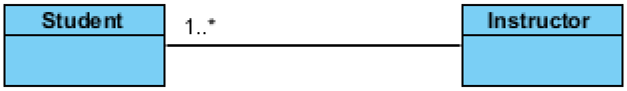
Association in Java defines the connection between two classes that are set up through their objects. Association manages **one-to-one,** **one-to-many**, and **many-to-**

many relationships. In Java, the multiplicity between objects is defined by the **Association**. It shows how objects communicate with each other and how they use the functionality and services provided by that communicated object. Association manages **one-to-one, one-to-many, many-to-one** and **many-to-many** relationships.

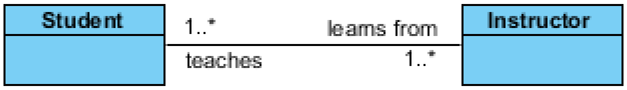


Let's take an example of each relationship to manage by the Association.

1. A person can have only one passport. It defines the **one-to-one**
2. If we talk about the Association between a College and Student, a College can have many students. It defines the **one-to-many**
3. A state can have several cities, and those cities are related to that single state. It defines the **many-to-one**
4. A single student can associate with multiple teachers, and multiple students can also be associated with a single teacher. Both are created or deleted independently, so it defines the **many-to-many**
5. A single student can associate with multiple teachers:
6. 

7. The example indicates that every Instructor has one or more Students:
8. 

9. We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.

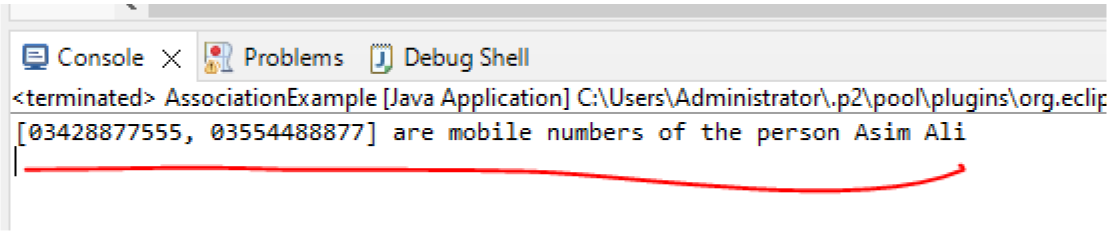
10. 

Let's takes an example of an Association to understand how it works in Java.

AssociationExample.java

```
package Lab5;
import java.util.*;
class Mobile {
    private String mobileNo;

    public String getMobileNo() {
        return mobileNo;
    }
    public void setMobileNo(String mobileNo) {
        this.mobileNo = mobileNo;
    }
    @Override
    public String toString() {
        return mobileNo;
    }
}
class Person {
    private String name;
    List<Mobile> numbers;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public List<Mobile> getNumbers() {
        return numbers;
    }
    public void setNumbers(List<Mobile> numbers) {
        this.numbers = numbers;
    }
}


public class AssociationExample {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
        Person person = new Person();
        person.setName("Asim Ali");

       Mobile number1 = new Mobile();
        number1.setMobileNo("03428877555");
        Mobile number2 = new Mobile();
        number2.setMobileNo("03554488877");

        List<Mobile> numberList = new ArrayList<Mobile>();
        numberList.add(number1);
        numberList.add(number2);
        person.setNumbers(numberList);

        System.out.println(person.getNumbers()+" are mobile numbers of the person "+
        person.getName());

         }
}
```

Output:



```
<terminated> AssociationExample [Java Application] C:\Users\Administrator\.p2\pool\plugins\org.eclip
[03428877555, 03554488877] are mobile numbers of the person Asim Ali
```

Description

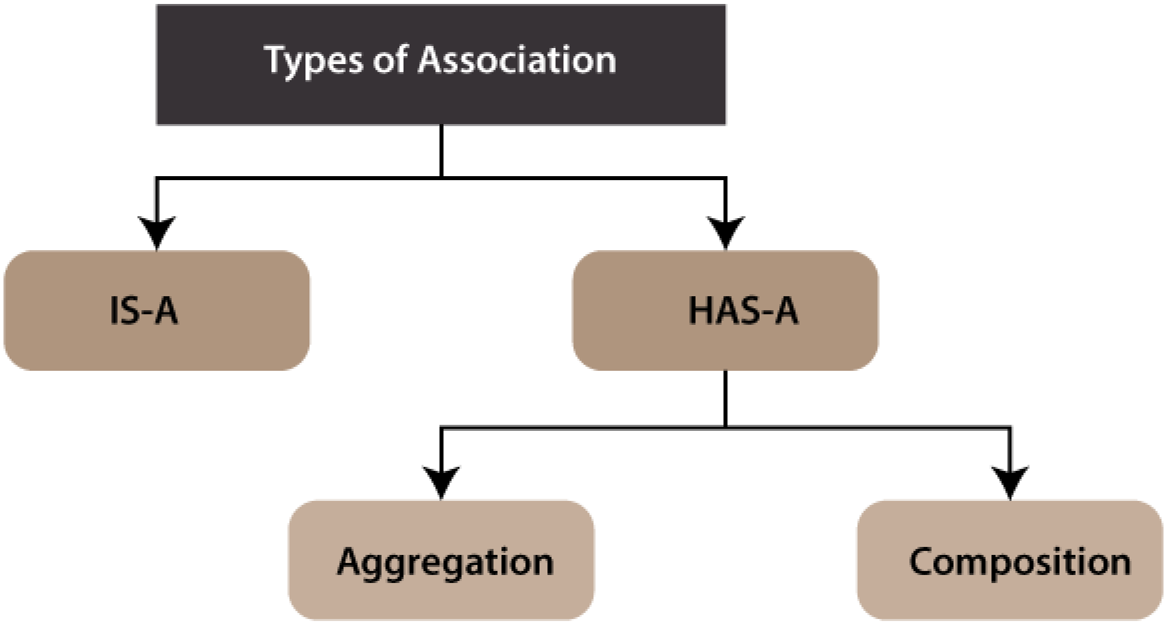
In the above example, we create two separate classes, i.e., Person and Mobile, associated through their objects. A person can have many mobile numbers, so it defines the one-to-many relationship.

Types of Association

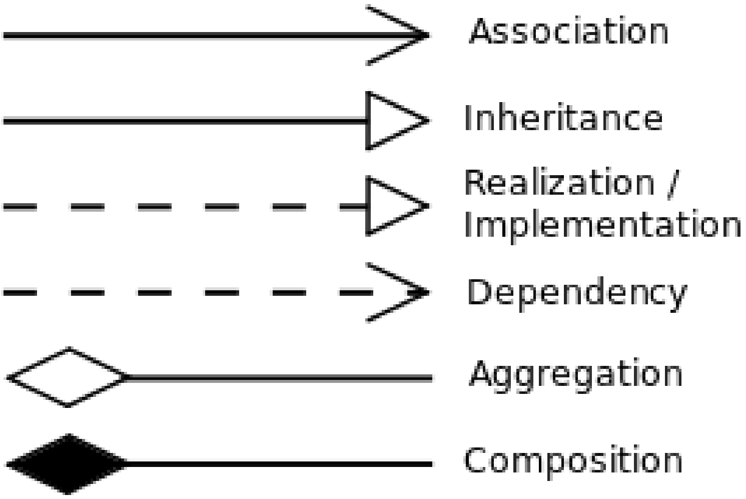
In Java, two types of **Association** are possible:

1. IS-A Association

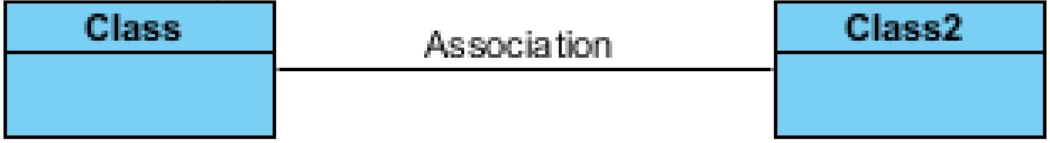
2. HAS-A Association
    a. Aggregation
    b. Composition

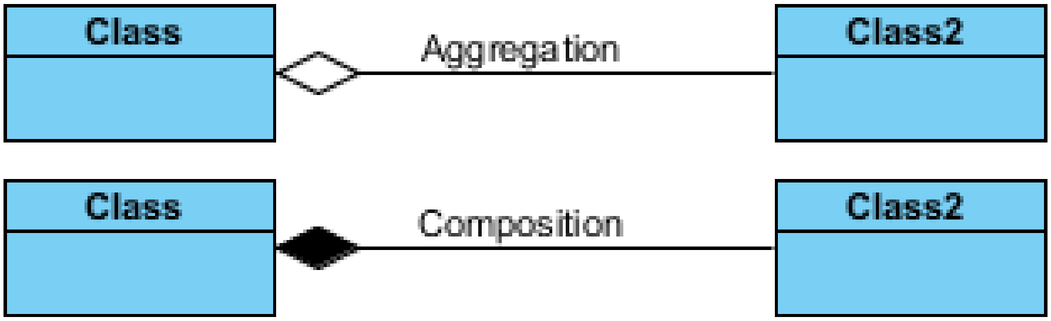


\

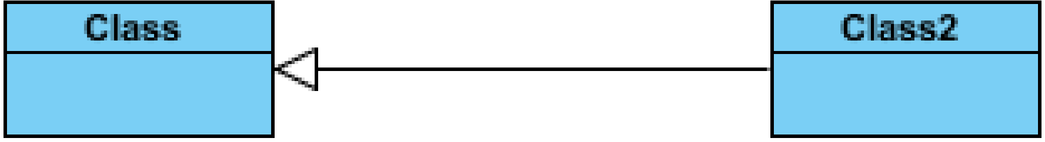


The figure below shows the three types of association connectors: association, aggregation, and composition.





The figure below shows a generalization. We will talk about it later on in this UML guide.

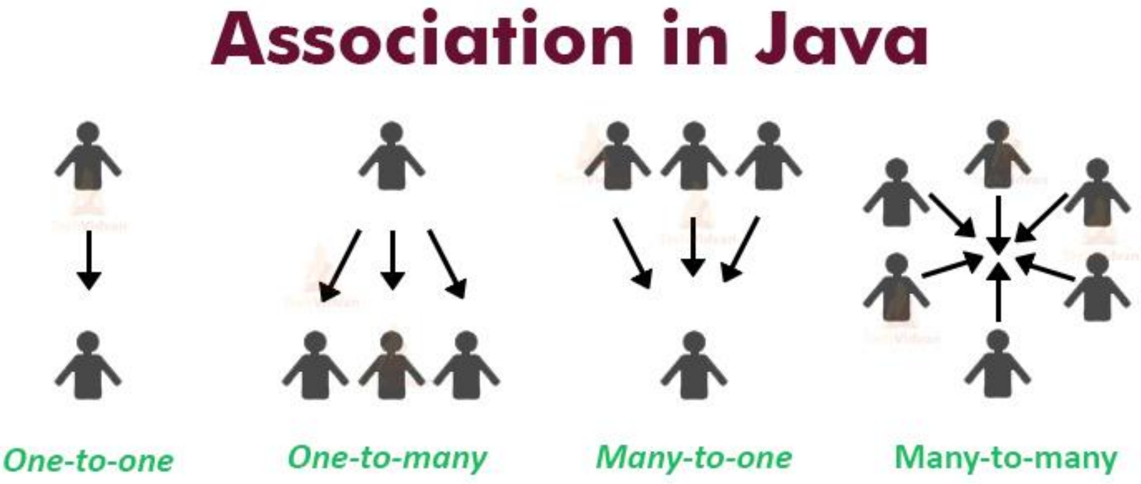


1) IS-A Association

The IS-A Association is also referred to as Inheritance. We all know about Inheritance in Java and if you don't know about it,

2) HAS-A Association

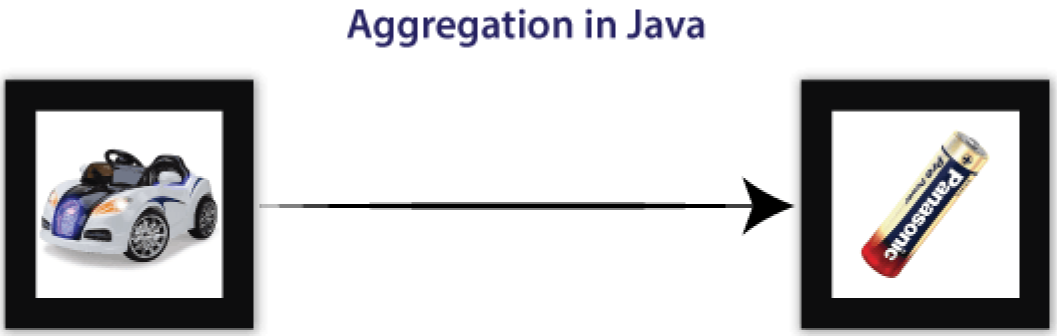
The **HAS-A Association** is further classified into two parts, i.e., Aggregation and Composition. Let's understand the difference between both of them one by one.



1) Aggregation

In Java, the **Aggregation** association defines the **HAS-A** relationship. Aggregation follows the one-to-one or one-way relationship. If two entities are in the aggregation association, and one entity fails due to some error, it will not affect the other entity.

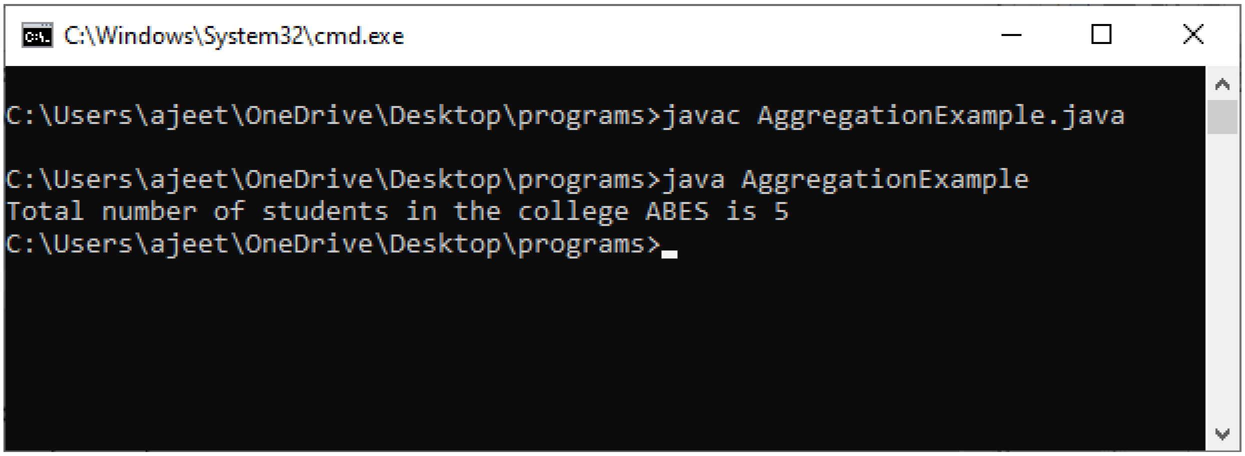
Let's take the example of a toy and its battery. The battery belongs to a toy, and if the toy breaks and deletes from our database, the battery will still remaining in our database, and it may still be working. So in Aggregation, objects always have their own lifecycles when the ownership exists there.



AggregationExample

```
import java.util.*;

// Student class
class Student
{
    String name;
    int enrol ;
    String course;

    Student(String name, int enrol, String course)
    {

        this.name = name;
        this.enrol = enrol;
        this.course = course;
    }
}

// Course class having a list of students.
class course
{

    String name;
    private List<Student> students;
    Course(String name, List<Student> students)
    {

        this.name = name;
        this.students = students;

    }

    public List<Student> studentsData()
    {
        return students;
    }
}

/* College class having a list of Courses*/
class College
{

    String collegeName;
    private List<Course> courses;

    College(String collegeName, List<Course> courses)
    {
        this.collegeName = collegeName;
        this.courses = courses;
    }

    // Returning number of students available in all courses in a given college
    public int countStudents()
    {
        int studentsInCollege = 0;
        List<Student> students;
        for(Course course : courses)
        {
            students = course.studentsData();
            for(Student s : students)
            {
                studentsInCollege++;
            }
        }
        return studentsInCollege;
    }

}

// main method
class AggregationExample
{
    public static void main (String[] args)
    {
        Student std1 = new Student("Ajnum", 1801, " BSC-SE ");
        Student std2 = new Student("Asif", 1802, "BSC-CS");
        Student std3 = new Student("Aslam", 1803, "EE");
        Student std4 = new Student("Amjid", 1804, "MCA");
        Student std5 = new Student("Kashif", 1805, "Poly");

        // Constructing list of MCA Students.
        List <Student> mca_students = new ArrayList<Student>();
        mca_students.add(std1);
        mca_students.add(std4);

        //Constructing list of BSC-CS Students.
        List <Student> bsc_cs_students = new ArrayList<Student>();
        bsc_cs_students.add(std2);

        //Constructing list of Poly Students.
        List <Student> poly_students = new ArrayList<Student>();
        poly_students.add(std3);
        poly_students.add(std5);

        Course MCA = new Course("MCA", mca_students);
        Course BSC_CS = new Course("BSC-CS", bsc_cs_students);
        Course Poly = new Course("Poly", poly_students);

        List <Course> courses = new ArrayList<Course>();
                courses.add(MCA);
                courses.add(BSC_CS);
                courses.add(Poly);

                // creating object of College.
                College college = new College("ABES", courses);

                System.out.print("Total number of students in the college "+ college.collegeName +" is "+ college.countStudents());
            }
        }
```

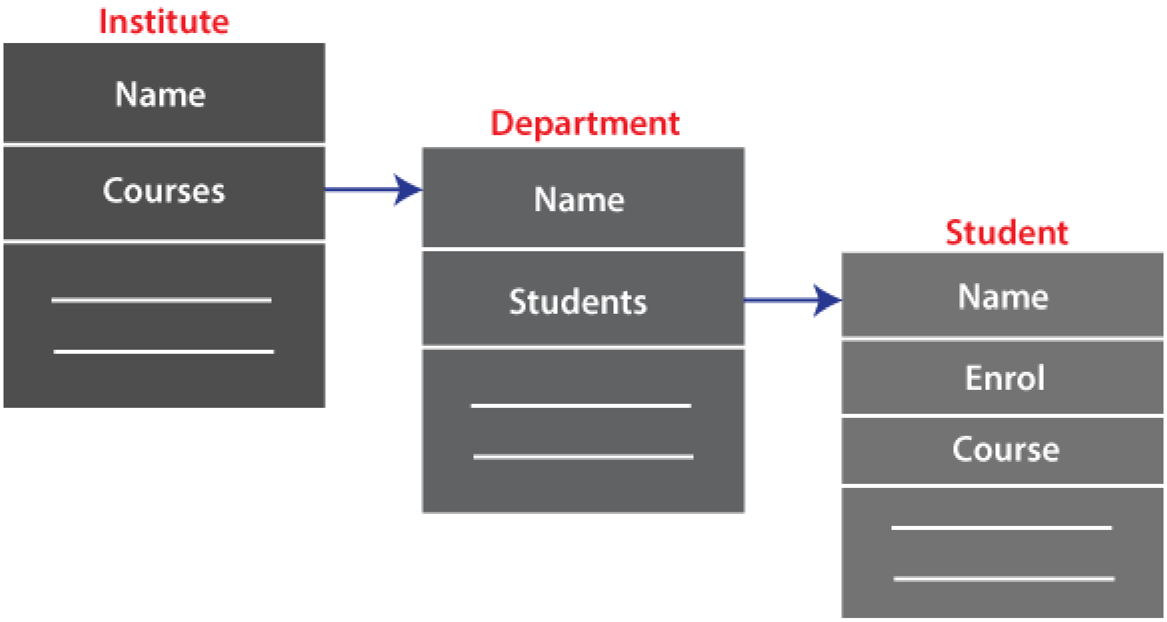
Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac AggregationExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java AggregationExample
Total number of students in the college ABES is 5
C:\Users\ajeet\OneDrive\Desktop\programs>_
```

Description:

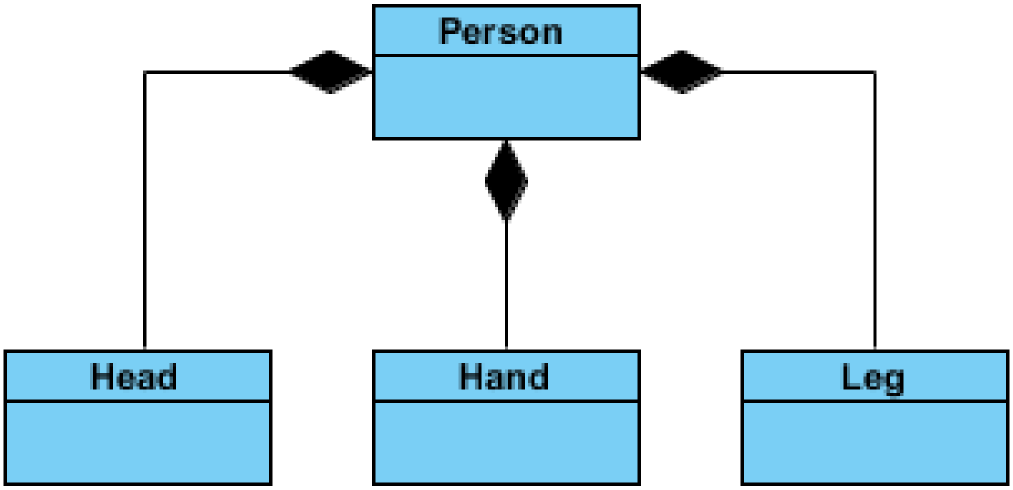
In the above example, there is a college that has several courses like BSC-CS, MCA, and Poly. Every course has several students, so we make a College class that has a reference to the object or list of objects of the Course class. That means College class is associated with Course class through the objects. Course class also has a reference to the object or list of objects of Student class means it is associated with Student class through its object and defines the **HAS-A** relationship.



2) Composition

A restricted form of the **Aggregation** where the entities are strongly dependent on each other. Unlike Aggregation, Composition represents the **part-of** relationship. When there is an aggregation between two entities, the aggregate object can exist without the other entity, but in the case of Composition, the composed object can't exist. To learn more about Composition,

We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result



Let's take an example to understand the concept of **Composition**.

We create a class **Mobile** that contains variables, i.e., **name, ram** and **rom**. We also create a class **MobileStore** that has a reference to refer to the list of mobiles. A mobile store can have more than one mobile. So, if a mobile store is destroyed, then all mobiles within that particular mobile store will also be destroyed because mobiles cannot exist without a mobile store. The relationship between the mobile store and mobiles is Composition.

CompositionExample.java:

```
import java.util.*;
class Mobile
{
  public String name;
  public String ram;
  public String rom;
  Mobile(String Name, String ram, String rom
```

```java
  {
    this.name = name;
    this.ram = ram;
    this.rom = rom;
  }
}
class MobileStore
{
  private final List<Mobile> mobiles;
  MobileStore (List<Mobile> mobiles)
  {
    this.mobiles = mobiles;
  }
  public List<Mobile> getTotalMobileInStore(){
    return mobiles;
  }
}
public class CompositionExample {
  public static void main (String[] args)
  {
    Mobile mob1 = new Mobile("Realme6","8GB", "128GB");
    Mobile mob2 = new Mobile("SAMSUNG A21S", "4GB", "128");
    Mobile mob3 = new Mobile("SAMSUNG M10", "4GB", "64GB");
    List<Mobile> mobiles = new ArrayList<Mobile>();
    mobiles.add(mob1);
    mobiles.add(mob2);
    mobiles.add(mob3);
    MobileStore store = new MobileStore(mobiles);
    List<Mobile> mob = store.getTotalMobileInStore();
    for(Mobile mb : mob){
      System.out.println("Name : " + mb.name + " RAM :" +mb.ram + " and "
          +" ROM : " + mb.rom);
    }
  }
```

```
}
```

Output:

Association Vs. Aggregation Vs. Composition

Association	Aggregation	Composition
Association relationship is denoted using an arrow.	Aggregation relationship is denoted using a straight line with an empty arrowhead at one end.	Composition relationship is denoted using a straight line with a filled arrowhead at any one of the ends.
Association can exist between two or more classes in UML.	Aggregation is a part of an association relationship.	The composition is a part of an association relationship.
There can be one-one, one-many, many-one, and many-many association present between the association classes.	Aggregation is considered as a weak type of association.	The composition is considered as a strong type of association.
In an association relationship, one or more objects can be associated with each other.	In an aggregation relationship, objects that are associated with each other can remain in the scope of a system without each other.	In a composition relationship, objects that are associated with each other cannot remain in the scope without each other.
Objects are linked with each other.	Linked objects are not dependent upon the other object.	Objects are highly dependent upon each other.
In UML Association, deleting one element may or may not affect another associated element.	In UML Aggregation, deleting one element does not affect another associated element.	In UML Composition, deleting one element affects another associated element.

Example: **A teacher is associated with multiple students.** **Or** **a teacher provides instructions to the students.**	Example: A car needs a wheel, but it doesn’t always require the same wheel. A car can function adequately with another wheel as well.	Example: A file is placed inside the folder. If one deletes the folder, then the file associated with that given folder is also deleted.