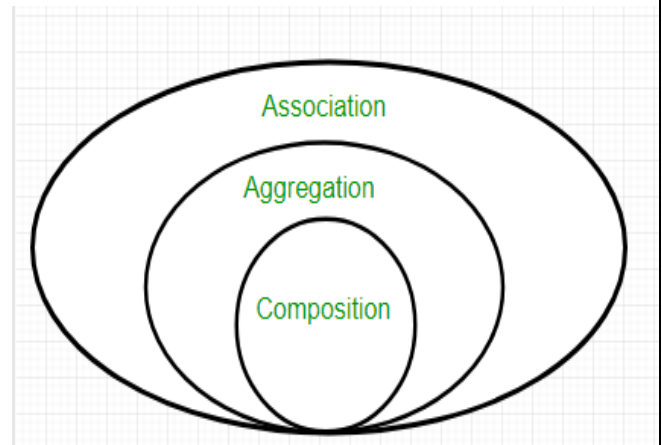


AGGREGATION AND COMPOSITION

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an **Object communicates to other Object** to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.



In the real world not only does **is-a** relationship exists also **has-a** relationship exists.

Is-a relationship is handled using **inheritance** whereas has-a relationship is handled using **aggregation and composition**.

Aggregation is a loose bound has-a relationship.
Composition is tight bound has-a relationship.

Let us now try to understand this in detail using few examples.

Example 1:

In the example shown below, a library contains students and books. Relationship between **library and student** is **aggregation** whereas relationship between **library and books** is **composition**.

A student can exist without library and therefore is aggregation. (**loosely bound**)

A book cannot exist without library and it is composition. (**tightly bound**)



Students



has-a
aggregation

Library



has-a
composition

Books

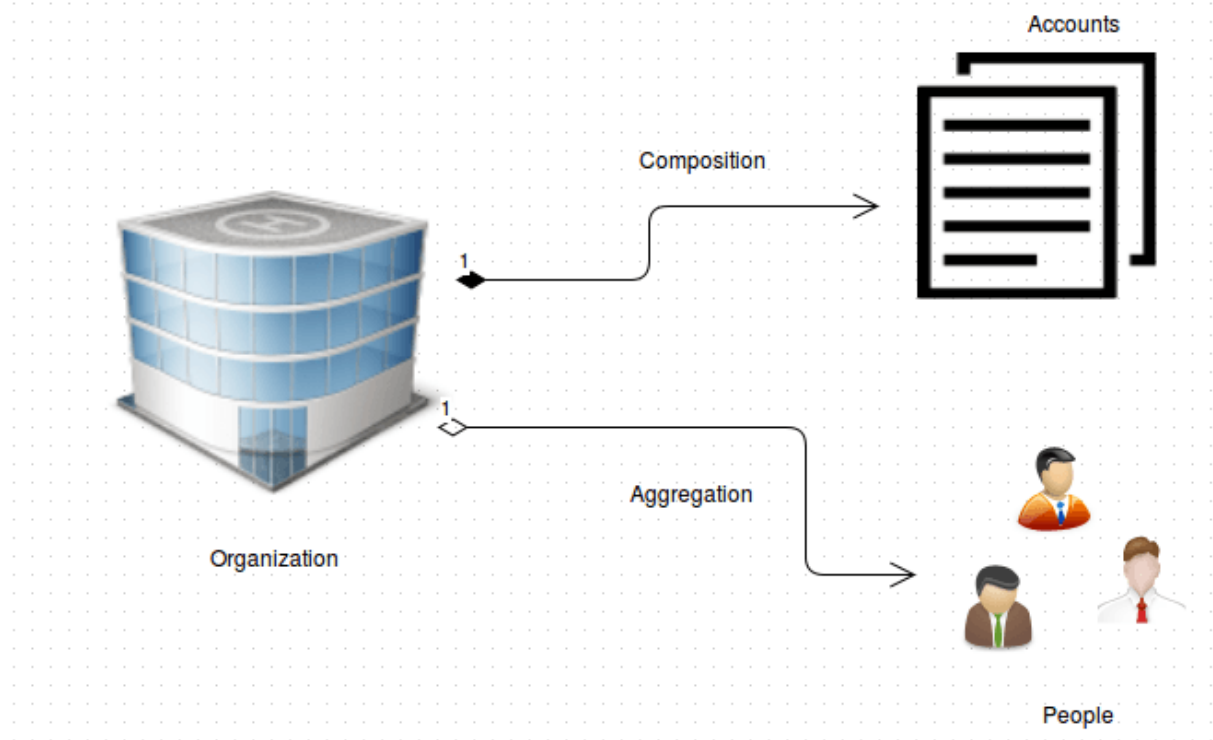


Example 2:

In this example shown below, an organization contains accounts and people. Relationship between **people and organization is aggregation** whereas relationship between **people and accounts is composition**.

People can exist without **organization** and therefore it is aggregation. (*loosely bound*)

Accounts cannot exist without **organization** and therefore it is composition. (*tightly bound*)



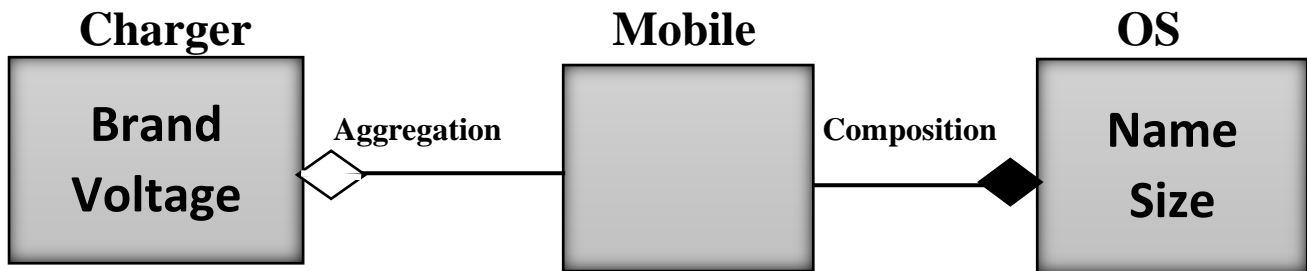
Example 3: In this example shown below, a phone contains an os and charger. Relationship between **phone and charger is aggregation** whereas relationship between **phone and os is composition**.

Charger can exist without **phone** and therefore it is aggregation. (*loosely bound*)

OS cannot exist without **phone** and therefore it is composition. (*tightly bound*)



Let us represent the third example in the form of UML diagram and start coding.



In the above UML Mobile has tight bound relationship with OS and loose bound relationship with charger. Charger has its own brand and voltage, similarly OS has its own name and size. Based on this information let us now start writing code.

CODE:

```
class Os
{
    private String name;
    private int size;
    Os(String name, int size)
    {
        this.name = name;
        this.size = size;
    }
    String getName()
    {
        return name;
    }
    int getSize()
    {
        return size;
    }
}
```

```
class Charger
{
    private String brand;
    private int voltage;
    Charger(String brand, int voltage)
    {
        this.brand = brand;
        this.voltage = voltage;
    }
    String getBrand()
    {
        return brand;
    }
    int getVoltage()
    {
        return voltage;
    }
}

class Mobile
{
    Os os = new Os("Android",512);
    void hasA(Charger c)
    {
        System.out.println(c.getBrand());
        System.out.println(c.getVoltage());
    }
}
```



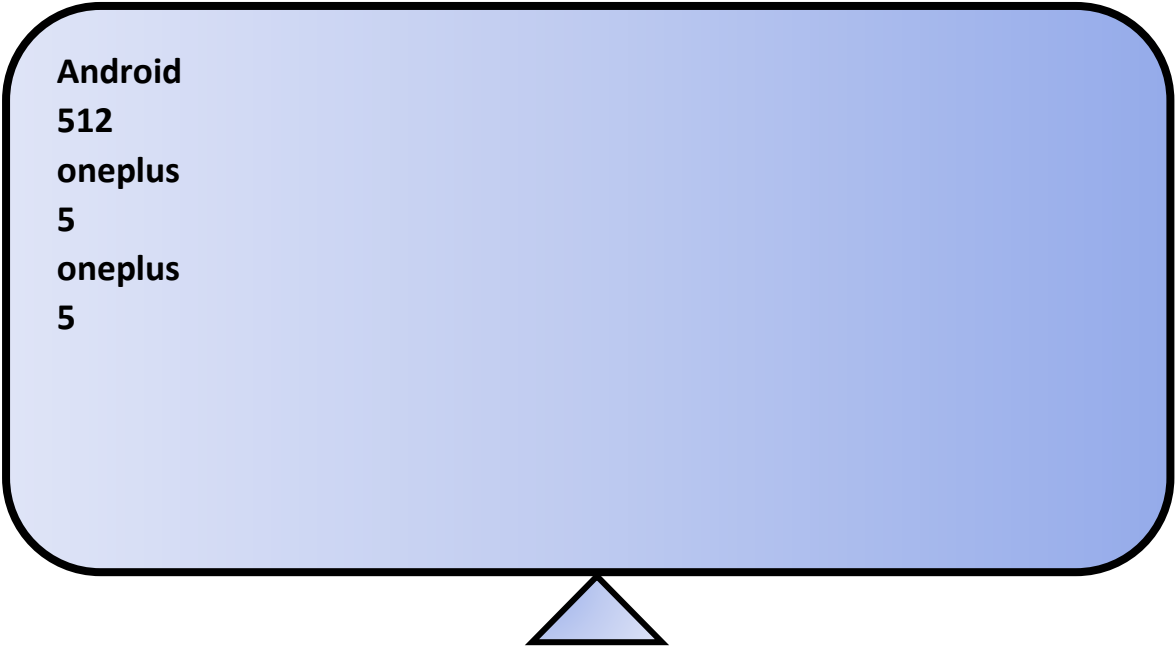
```

class Demo
{
    public static void main(String[] args)
    {
        Charger c = new Charger("oneplus",5);
        Mobile m = new Mobile();
        System.out.println(m.os.getName());
        System.out.println(m.os.getSize());
        m.hasA(c);
        m=null;
        //System.out.println(m.os.getName()); --->error
        //System.out.println(m.os.getSize()); --->error
        //m.hasA(c); --->error
        System.out.println(c.getBrand());
        System.out.println(c.getVoltage());
    }
}

```

In the above code if the **reference** to the **mobile** is made **null**, we **cannot access os** as os has **tight bound** relationship with mobile and if attempted we **get error**. But we can **access charger using charger type reference** as charger has **loose bound relationship with mobile**.

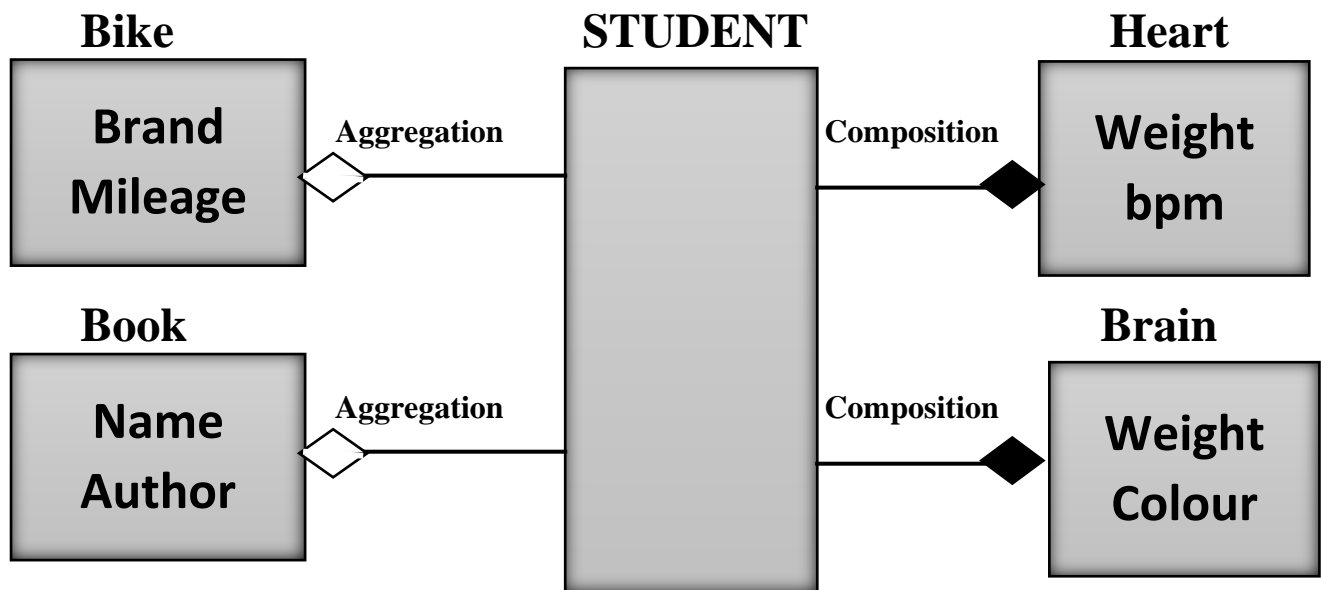
OUTPUT



```

Android
512
oneplus
5
oneplus
5

```



In the above UML, student has tight bound relationship with brain and heart, and loose bound relationship with book and bike.

Bike Consists of **brand** and **mileage**.

Book consists of **name** and **author**.

Heart consists of **weight** and **bpm**.

Brain consists of **weight** and **colour**.

Based on the above information let us now start writing code:

CODE:

```

class Heart
{
    private int weight;
    private int bpm;
    Heart(int weight, int bpm)
    {
        this.weight = weight;
        this.bpm = bpm;
    }
    int getWeight()
    {
        return weight;
    }
    int getBpm()
    {
        return bpm;
    }
}

```

```
class Brain
{
    private int weight;
    private String colour;
    Brain(int weight, String colour)
    {
        this.weight = weight;
        this.colour = colour;
    }
    int getWeight()
    {
        return weight;
    }
    String getColour()
    {
        return colour;
    }
}

class Book
{
    private String name;
    private String author;
    Book(String name, String author)
    {
        this.name = name;
        this.author = author;
    }
    String getName()
    {
        return name;
    }
    String getAuthor()
    {
        return author;
    }
}
```



```
class Bike
{
    private String brand;
    private int mileage;
    Bike(String brand, int mileage)
    {
        this.brand = brand;
        this.mileage = mileage;
    }
    String getBrand()
    {
        return brand;
    }
    int getMileage()
    {
        return mileage;
    }
}
```

```
class Student
{
    Heart h = new Heart(289,72);
    Brain b = new Brain(1400,"gray");
    void hasA(Book book)
    {
        System.out.println(book.getName());
        System.out.println(book.getAuthor());
    }
    void hasA(Bike bike)
    {
        System.out.println(bike.getBrand());
        System.out.println(bike.getMileage());
    }
}
```



```
class Demo
{
    public static void main(String[] args)
    {
        Student s = new Student();
        Bike bike = new Bike("Duke",35);
        Book book = new Book("Java","JG");
        System.out.println(s.h.getWeight());
        System.out.println(s.h.getBpm());
        System.out.println(s.b.getWeight());
        System.out.println(s.b.getColour());
        s.hasA(bike);
        s.hasA(book);
        s=null;

        //System.out.println(s.h.getWeight()); ---->error
        //System.out.println(s.h.getBpm()); ---->error
        //System.out.println(s.b.getWeight()); ---->error
        //System.out.println(s.b.getColour()); ---->error
        //s.hasA(bike); ---->error
        //s.hasA(book); ---->error

        System.out.println(bike.getBrand());
        System.out.println(bike.getMileage());
        System.out.println(book.getName());
        System.out.println(book.getAuthor());

    }
}
```

Can you guess the output????



OUTPUT

289
72
1400
gray
Duke
35
Java
JG
Duke
35
Java
JG



Difference between Aggregation and Composition:

Aggregation	Composition
Aggregation is a weak Association.	Composition is a strong Association.
Class can exist independently without owner.	Class can not meaningfully exist without owner.
Have their own Life Time.	Life Time depends on the Owner.
A uses B.	A owns B.
Child is not owned by 1 owner.	Child can have only 1 owner.
Has-A relationship. A has B.	Part-Of relationship. B is part of A.
Denoted by a empty diamond in UML.	Denoted by a filled diamond in UML.
We do not use "final" keyword for Aggregation.	"final" keyword is used to represent Composition.
Examples: <ul style="list-style-type: none">- Car has a Driver.- A Human uses Clothes.- A Company is an aggregation of People.- A Text Editor uses a File.- Mobile has a SIM Card.	Examples: <ul style="list-style-type: none">- Engine is a part of Car.- A Human owns the Heart.- A Company is a composition of Accounts.- A Text Editor owns a Buffer.- IMEI Number is a part of a Mobile.