Topic: Composition of Classes.
OOP concepts involved: Classes, Constructor Method, Object Instances, Association Relationships (Aggregation and Composition).
Programming generic concepts involved: Variables, Data Types, Functions, Access Modifiers.

➢ Theoric introduction

IMPORTANT RELATIONSHIPS IN JAVA

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 implementation 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 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.

HAS-A Relationship:
Composition (**HAS-A**) simply means the use of instance variables that are references to other objects. For example, a **Car** has **Engine**, or **House** has **Bathroom**.

Composition allows us to model objects that are made up of other objects, thus defining a **HAS-A** relationship between them.

ASSOCIATION IN OBJECT-ORIENTED PROGRAMMING

The association relationship indicates that a class knows about, and holds a reference to, another class. Association establishes a relationship between two separate **classes** through their objects. The relationship can be one to one, one to many, many to one and many to many. **Composition** and **aggregation** are two types of association.

Associations join one or more of one thing against one or more of another thing. A professor might be associated with a college course (a one-to-one relationship) but also with each student in her class (a one-to-many relationship). The students in one section might be associated with the students in another section of the same course (a many-to-many relationship) while all the sections of the course relate to a single course (a many-to-one relationship).

COMPOSITION

Composition is the strongest form of association, which means that the object(s) that compose or are contained by one object are destroyed too when that object is destroyed.

Composition is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For example Human and Heart. A human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition. Heart class has no sense if Human class is not present.

```
//Human has a Heart
public class Human{
      //Other Human attributes
      private final Heart heart; //Heart is a mandatory part of the Human

   public Human() {
      heart = new Heart();
   }
}

//Heart Object
class Heart{
```

```
// class members
}
```

AGGREGATION

Aggregation is a weak association. An association is said to be aggregation if the composed class (entity) can exist independently of the Parent entity (class) .

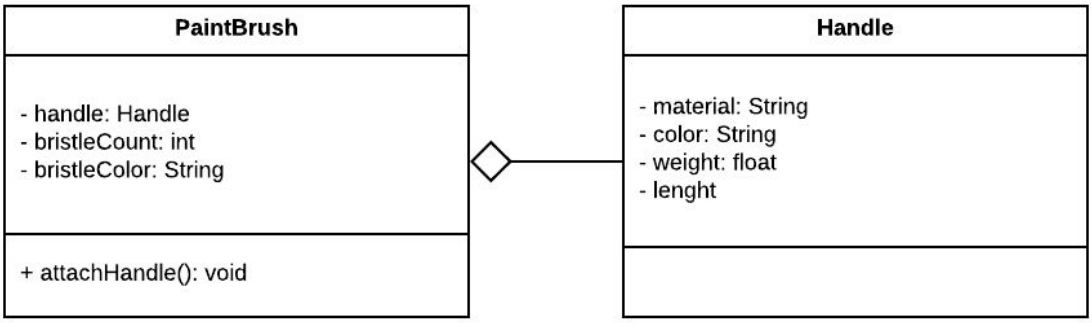
Aggregation is a special form of association which is a unidirectional one-way relationship between classes (or entities), for example, Wallet and Money classes. Wallet has Money but money doesn't need to have Wallet necessarily so it's one directional relationship. In our example, if Wallet class is not present, it does not mean that the Money class cannot exist.

```
//Wallet class
public class Wallet{
  // other Wallet attributes
   private Money money;

   public Wallet() {
      money = new Money();
   }
}

//Money class
class Money{
// class members
}
```

➢ Statement

Create a class called **Handle** that contains the attributes of *material* (String), *color* (String), *weight* (float) and *length* (float). After that, create a class called **PaintBrush** that contains an aggregation relationship, which consists of an attribute of type **Handle**, and two other attributes: *bristleCount* (int) and *bristleColor* (String). In turn, create a method called *attachHandle* to display a message that uses the attributes of the Handle class and the attributes of the PaintBrush class.

➢ Class design (UML)



➢ Program Code

Handle.java

```
public class Handle {
	// Handle attributes
	private String material;
	private String color;
	private float weight;	// grams
	private float length;	// cm

	public Handle(String material, String color, float weight, float length) {
		this.material = material;
		this.color = color;
		this.weight = weight;
		this.lenght = lenght;
	}
```

```
	public String getMaterial() {
		return material;
	}

	public void setMaterial(String material) {
		this.material = material;
	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	public float getWeight() {
		return weight;
	}

	public void setWeight(float weight) {
		this.weight = weight;
	}

	public float getLength() {
		return length;
	}

	public void setLength(float length) {
		this.lenght = lenght;
	}
}
```

PaintBrush.java

```
public class PaintBrush {
	private Handle handle; // The handle entity is the main part of the
aggregation
	private int bristleCount;
	private String bristleColor;

	// Constructor
	public PaintBrush(Handle handle, int bristleCount, String bristleColor) {
		this.handle = handle;
```

```
		this.bristleCount = bristleCount;
		this.bristleColor = bristleColor;
	}

	// own PaintBrush method
	public void attachHandle() {
		System.out.println("The " + handle.getColor() + " " +
handle.getMaterial() + " Handle has been attached to \n the "+ bristleColor + "
paint brush with " + bristleCount + " bristles");
	}

	// Getters and Setters
	public Handle getHandle() {
		return handle;
	}

	public void setHandle(Handle handle) {
		this.handle = handle;
	}

	public int getBristleCount() {
		return bristleCount;
	}

	public void setBristleCount(int bristleCount) {
		this.bristleCount = bristleCount;
	}

	public String getBristleColor() {
		return bristleColor;
	}

	public void setBristleColor(String bristleColor) {
		this.bristleColor = bristleColor;
	}
}
```

Main.java

```
public class Main {
	public static void main(String[] args) {
		Handle handle = new Handle("Wood", "Brown", 30, 12);

		PaintBrush paintBrush = new PaintBrush(handle, 1000, "White");
		paintBrush.attachHandle();
	}
```

```
}
```

➢ **Program execution**

In this simple program, the instance of a class that has an aggregation relationship corresponding to the **Handle** class was created.

First, the instance of the Handle object was created, which passed parameters to call the constructor. After this, a PaintBrush type object was created in which the Handle object was passed through parameters, along with *bristleCount* and *bristleColor*. Finishing, the *attachHandle* method was called.

```
The Brown Wood Handle has been attached to
 the White paint brush with 1000 bristles
```

➢ **Conclusions**

There are two divisions of Association: Aggregation which represents a uni-directional weaker association between entities, and Composition which represents a stronger association between entities, where one entity cannot exist without the other.
We use Aggregation for code reusability, for having the ability to create a class for a specific object and using that object in other entities for its composement.

We know that Composition is a restricted form of Aggregation in which two entities are highly dependent on each other. When there is a composition between two entities, the composed object cannot exist without the other entity.