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) .

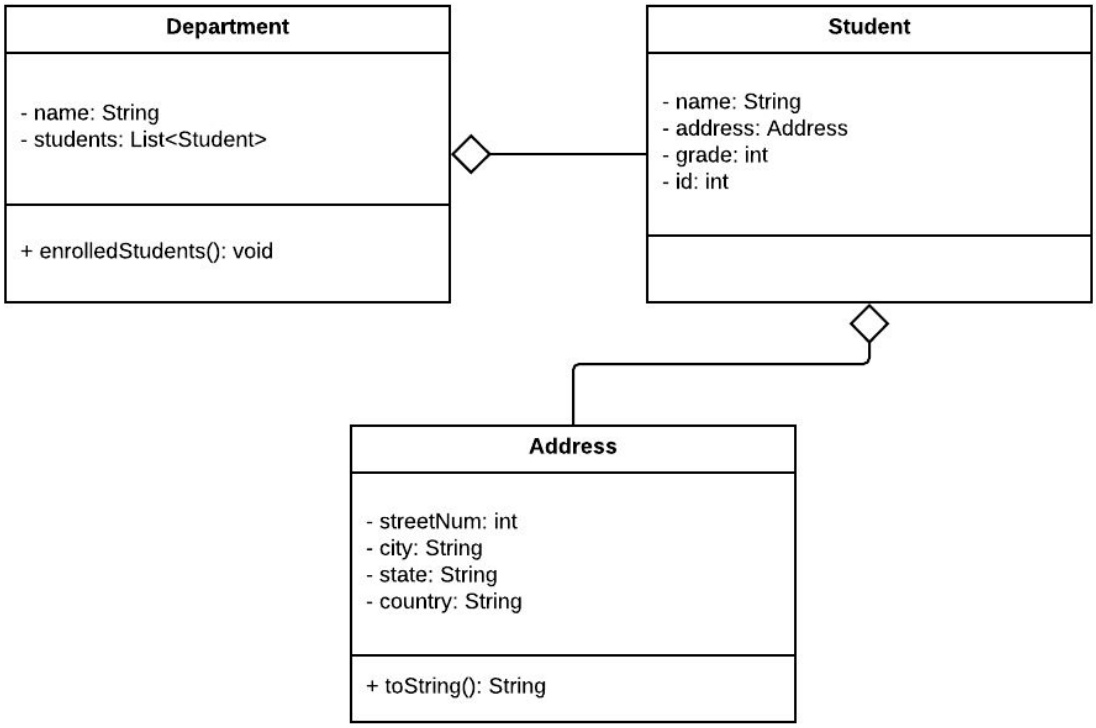
Aggregation is a special form of association which is a unidirectional one-way relationship between classes (or entities), for example, Wallet and Money classes. Wallet has Money but money doesn't need to have Wallet necessarily so it's one directional relationship. In our example, if Wallet class is 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 named **Address** that contains the attributes of *streetNum* (int), *city* (String), *state* (String) and *country* (String). After that, create a class named **Student** that contains an aggregation relationship, which consists of an attribute of type **Address**, and three other attributes: *name* (String), *grade* (int) and *id* (int). Once this is finished, create a class named **Department** which consists of an aggregation relation where its attributes are: *students* (List<Student>) and *name* (String). Also, create a method *enrolledStudent*, same that prints out on screen a list of the enrolled students.

➢ Class design (UML)



➢ **Program Code**

Address.java

```
public class Address {
	private int streetNum;
	private String city;
	private String state;
	private String country;

	public Address(int streetNum, String city, String state, String country) {
		this.streetNum = streetNum;
		this.city = city;
		this.state = state;
		this.country = country;
	}

	@Override
	public String toString() {
		return "Address [Street No. = " + streetNum + ", City = " + city + ",
State = " + state + ", Country = " + country + "]";
	}

	public int getStreetNum() {
		return streetNum;
	}

	public void setStreetNum(int streetNum) {
		this.streetNum = streetNum;
	}

	public String getCity() {
		return city;
	}

	public void setCity(String city) {
		this.city = city;
	}

	public String getState() {
		return state;
	}

	public void setState(String state) {
		this.state = state;
	}
```

```
	public String getCountry() {
		return country;
	}

	public void setCountry(String country) {
		this.country = country;
	}
}
```

Student.java

```
public class Student {
	// Student attributes
	private String name;
	private Address address;  // Address entity for the aggregation
	private int grade;
	private int id;

	public Student(String name, Address address, int grade, int id) {
		this.name = name;
		this.address = address;
		this.grade = grade;
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public Address getAddress() {
		return address;
	}

	public void setAddress(Address address) {
		this.address = address;
	}

	public int getGrade() {
		return grade;
	}
```

```
	public void setGrade(int grade) {
		this.grade = grade;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}
}
```

Department.java

```
public class Department {
	private String name;
	private List<Student> students; // List<Student> entity for the Aggregation

	public Department(String name, List<Student> students) {
		this.name = name;
		this.students = students;
	}

	public void enrolledStudents() {
		for (Student stud : students) {
			System.out.println("ID No: "+stud.getId() + " - " + "Student
name: " + stud.getName() + ", Student grade: "
					+ stud.getGrade() + "\n " +
stud.getAddress().toString()+"\n");
		}
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

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

```
	public void setStudents(List<Student> students) {
		this.students = students;
	}
}
```

Main.java

```
public class Main {
	public static void main(String[] args) {
		// Creating instances of Address class for two students
		Address studentAdr1 = new Address(123, "Tijuana", "Baja California",
"Mexico");
		Address studentAdr2 = new Address(5123, "San Diego", "California",
"United States");

		// Creating instances of Student class
		Student student1 = new Student("Ernesto Flores", studentAdr1, 9,
74123);
		Student student2 = new Student("James Dolg", studentAdr2, 12, 74281);

		// Grouping all the students for adding them to the math department
		List<Student> students = new ArrayList<Student>();
		students.add(student1);
		students.add(student2);

		Department depMath = new Department("Advanced Math",students);
		depMath.enrolledStudents();

	}
}
```

➢ Program execution

First, two instances of the **Address** object were created, which passed parameters to call the constructor. After this, two **Student** type object was created in which the respective **Address** object was passed through parameters, along with *name*, *grade,* and *id*. Once both Student objects were created, a List of Student was created, adding after that both students (student1 and student2) to the List.

In the end, a **Department** object was created which received as parameters a List of Student object and a name (String). Afterward, the method *enrolledStudents* were called, where all the students of the Department were shown.

```
ID No: 74123 - Student name: Ernesto Flores, Student grade: 9
 Address [Street No. = 123, City = Tijuana, State = Baja California, Country = Mexico]

ID No: 74281 - Student name: James Dolg, Student grade: 12
 Address [Street No. = 5123, City = San Diego, State = California, Country = United States]
```

➢ 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.