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) .

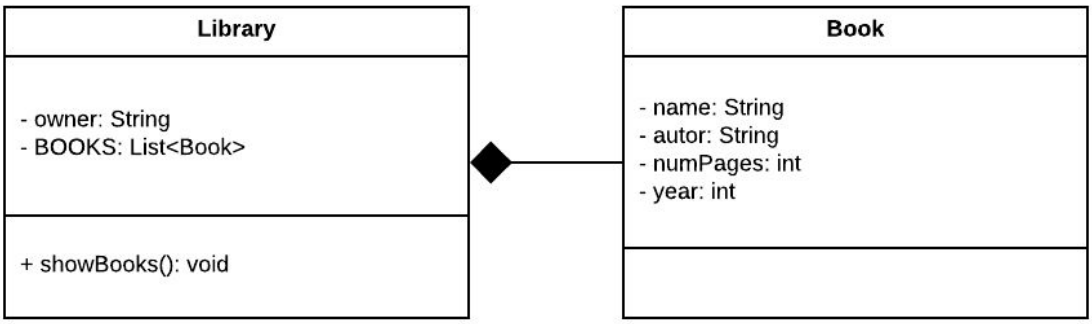
Aggregation is a special form of association which is a unidirectional one-way relationship between classes (or entities), for example, Wallet and Money classes. Wallet has Money but money doesn't need to have Wallet necessarily so it's one directional relationship. In our example, if Wallet class is 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 **Book** that contains the attributes of *name* (String), *author* (String), *numPages* (int) and *year* (int). After that, create a class called **Library** that contains an *owner* (String) attribute as well as a composition relationship, which consists of an attribute of type **List<Book>**. In turn, create a method called *showBooks* to display all the Books that are available within the Library.

➢ Class design (UML)



➢ Program Code

Book.java

```
public class Book {
	private String name;
	private String author;
	private int numPages;
	private int year;

	public Book(String name, String author, int numPages, int year) {
		this.name = name;
		this.author = autor;
		this.numPages = numPages;
		this.year = year;
	}

	public String getName() {
```

```
		return name;
	}

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

	public String getAuthor() {
		return author;
	}

	public void setAuthor(String autor) {
		this.author = author;
	}

	public int getNumPages() {
		return numPages;
	}

	public void setNumPages(int numPages) {
		this.numPages = numPages;
	}

	public int getYear() {
		return year;
	}

	public void setYear(int year) {
		this.year = year;
	}
}
```

Library.java

```
public class Library {
	// Library attributes
	private String owner;
	private final List<Book> books; // final because a library can only have one
reference to a List of books

	public Library(String owner, List<Book> books) {
		this.owner = owner;
		this.books = books;
	}
```

```
	public void showBooks() {
		System.out.println(owner+" library has:");
		for (Book book : books) {
			System.out.println("Book Name: " + book.getName() + ", Author:
" + book.getAuthor() + ",\n Year: "
					+ book.getYear() + ", No. of pages: " +
book.getNumPages() + "\n");
		}
	}

	// Getters and Setters
	public String getOwner() {
		return owner;
	}

	public void setOwner(String owner) {
		this.owner = owner;
	}

	public List<Book> getBooks() {
		return books;
	}

}
```

Main.java

```
public class Main {

	public static void main(String[] args) {
		// Creating an instance of a couple of Books
		Book book1 = new Book("Harry Potter and the Philosopher's Stone",
"J.K. Rowling", 256, 1997);
		Book book2 = new Book("Don Quixote", "Miguel de Cervantes", 928,
1620);

		// Adding books to a list of Book
		List<Book> books = new ArrayList<Book>();
		books.add(book1);
		books.add(book2);

		// Passing the list of books and the owner to the Library constructor
		Library library = new Library("Juan Riles", books);
		library.showBooks();
	}
```

```
}
```

➢ Program execution

In this simple program, the instance of a class that has a composition relationship corresponding to the **Book** class was created.

First, two instances of the *Book* class were created, which passed parameters to call the constructor. After this, a variable of type *List<Book>* was created, where the two *Book* instances were added to. Afterward, a *Library* type object was created in which the *List<Book>* object was passed through parameters, along with the name of the Library *owner*. Finishing, the *showBooks* method was called, for showing all *Book* type objects stored in the Library.

```
Juan Riles library has:
Book Name: Harry Potter and the Philosopher's Stone, Autor: J.K. Rowling,
 Year: 1997, No. of pages: 256

Book Name: Don Quixote, Autor: Miguel de Cervantes,
 Year: 1620, No. of pages: 928
```

➢ 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.