OOC-1 Recap

Scope of an Object

- **Local Scope**: Objects with local scope are defined within a specific block or method.
- **Instance Scope**: Objects with instance scope are created as part of a class and are associated with instances (objects) of that class.
- **Static Scope (Class Scope)**: Objects or variables with static scope belong to the class itself rather than to instances of the class.
- **Method Parameter Scope**: Objects passed as method parameters have a scope within that method. They exist only for the duration of that method's execution.
- **Block Scope**: Objects declared within a block of code, such as loops or conditional statements, have a scope limited to that block.
- **Method Return Scope**: Objects returned from a method exist in the scope of the caller. They are typically used to pass data back to the caller.

```
void someMethod() {
    int localVar = 10; // localVar has local scope
    // ...
}
```

```
class MyClass {
    int instanceVar; // instanceVar has instance scope
    void instanceMethod() {
        // instanceVar can be accessed here
    }
}
```

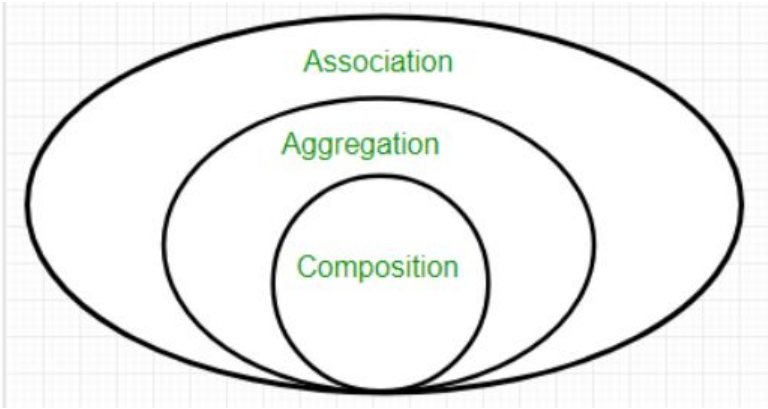
```
class MyClass {
    static int staticVar; // staticVar has static scope
    static void staticMethod() {
        // staticVar can be accessed here
    }
}
```

```
void someMethod(Object parameterObj) {
    // parameterObj can be accessed here
}
```

```
void someMethod() {
    if (condition) {
        int blockVar = 5; // blockVar has block scope
        // ...
    }
    // blockVar is not accessible here
}
```

```
Object someMethod() {
    Object resultObj = // ...
    return resultObj;
}
void anotherMethod() {
    Object returnedObj = someMethod();
    // returnedObj can be accessed here
}
```

Association, 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.

Example

```java
class Address
{
    int streetNum;
    String city;
    String state;
    String country;
    Address(int street, String c, String st, String coun)
    {
        this.streetNum=street;
        this.city =c;
        this.state = st;
        this.country = coun;
    }
}
```

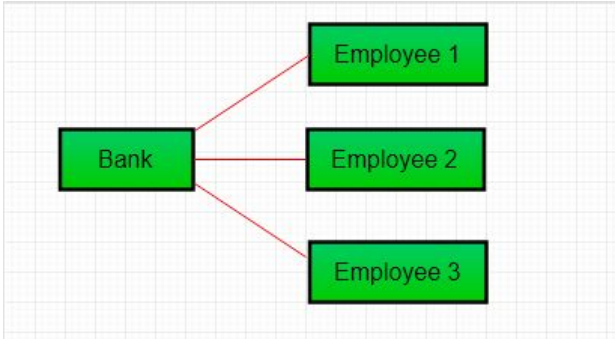
```java
class Student
{
    int rollNum;
    String studentName;

    Address studentAddr; //Creating HAS-A relationship with Address class

    Student(int roll, String name, Address addr){
        this.rollNum=roll;
        this.studentName=name;
        this.studentAddr = addr;
    }
}
```

```
Student Has-A Address (Has-a relationship between student and address)
College Has-A Address (Has-a relationship between college and address)
Staff Has-A Address (Has-a relationship between staff and address)
```

Association

```java
class Bank
{
    private String name;
    Bank(String name){
        this.name = name;
    }
    public String getBankName(){
        return this.name;
    }
}
```

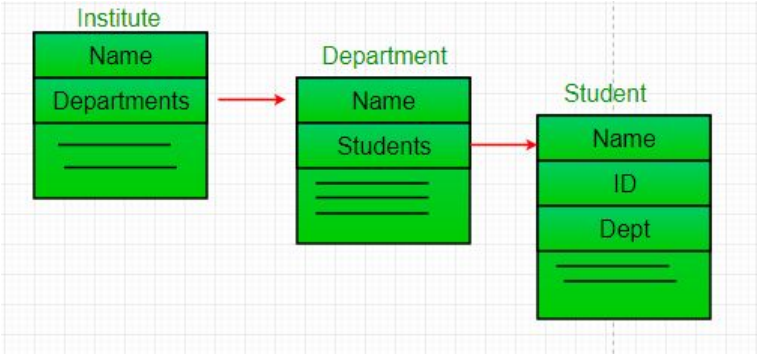
```java
class Employee
{
    private String name;
    Employee(String name){
        this.name = name;
    }
    public String getEmployeeName(){
        return this.name;
    }
}
```




```java
class Association
{
    public static void main (String[] args)
    {
        Bank bank = new Bank("Bangladesh Bank");
        Employee emp = new Employee("Reza");

        System.out.println(emp.getEmployeeName() +
                " is employee of " + bank.getBankName());
    }
}
```

Aggregation

- It is a special form of Association.
- It represents **Has-A** relationship.
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity



Aggregation

```
class Student
{
    String name;
    int id ;
    String dept;

    Student(String name, int id, String dept){
        this.name = name;
        this.id = id;
        this.dept = dept;
    }
}
```

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

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

```
public static void main (String[] args)
{
    Student s1 = new Student("Bob", 1, "CSE");
    Student s2 = new Student("Alice", 2, "EE");
    Student s3 = new Student("Ned", 2, "CSE");
    Student s4 = new Student("Jon", 1, "EE");

    List <Student> cse_students = new ArrayList<Student>();
    cse_students.add(s1);
    cse_students.add(s3);

    List <Student> ee_students = new ArrayList<Student>();
    ee_students.add(s2);
    ee_students.add(s4);

    Department CSE = new Department("CSE", cse_students);
    Department EE = new Department("EE", ee_students);
```

Composition

- Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.
- It represents **part-of** relationship.
- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.
- **Ex:** House and room, Library and Book

Composition

- It is natural to think of objects as containing other objects.
- A computer contains video cards, keyboards, and drives. Although the computer can be considered an object unto itself, the drive is also considered a valid object.
  - you could open up the computer and remove the drive
- a television *has-a* tuner and *has-a* video display
- In this way, objects are often built, or composed, from other objects: This is composition.
- Whenever a particular object is composed of other objects, and those objects are included as object fields, the new object is known as a *compound*, an *aggregate*, or a *composite object*

Composition

- Library and Book

```java
class Book
{
    public String title;
    public String author;

    public Book(String title, String author){
        this.title = title;
        this.author = author;
    }
}
```

```java
class Library
{
    private final List<Book> books;
    public Library (List<Book> books){
        this.books = books;
    }
    public List<Book> getTotalBooksInLibrary(){
        return books;
    }
}
```

```java
public static void main (String[] args)
{
    Book b1 = new Book("Java: The Complete Reference", "Herbert Schildt");
    List<Book> books = new ArrayList<Book>();
    books.add(b1);

    Library library = new Library(books);

    List<Book> bks = library.getTotalBooksInLibrary();
    for(Book bk : bks){
        System.out.println("Title : " + bk.title + " and "
        +" Author : " + bk.author);
    }
}
```

Composition-Aggregation

Simple Rule

A "owns" B = Composition: B has no meaning or purpose in the system without A

A "uses" B = Aggregation: B exists independently (conceptually) from A

Example 1:

A Company is an aggregation of People. A Company is a composition of Accounts. When a Company ceases to do business its Accounts cease to exist but its People continue to exist.

Example 2: (very simplified)

A Text Editor owns a Buffer (composition). A Text Editor uses a File (aggregation). When the Text Editor is closed, the Buffer is destroyed but the File itself is not destroyed.

Aggregation vs Composition

- **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
- **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”** relation.
- **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

Delegation

- **Delegation** means hand over the responsibility for a particular task to another class or method.
- It is a technique where an object expresses certain behavior to the outside but in reality delegates responsibility for implementing that behavior to an associated object.
- Which class actually perform the responsibility is called delegate.
- Which class pass the responsibility to other classes is called delegator.

Delegation

- Delegation: Your boss asked you to get him a coffee, you've had an intern do it for you instead.
- **Delegation:** When my object uses another object's functionality as is without changing it.

- Example: Ticket Booking, Printer

```
public class A {
  private B b = new B();

  public void methodA() {
    b.methodB();
  }
}
```

Example: Ticket booking

```
interface TravelBooking {
    public void bookTicket();
}
```

```
class TrainBooking implements TravelBooking {
    @Override
    public void bookTicket() {
        System.out.println("Train ticket booked");
    }
}
```

```
class AirBooking implements TravelBooking {
    @Override
    public void bookTicket() {
        System.out.println("Flight ticket booked");
    }
}
```

```
class TicketBookingAgent{
    TravelBooking t;
    public TicketBookingAgent(TravelBooking t){
        this.t = t;
    }

    // Ticket booking is delegated to other class using polymorphism
    public void bookTicket(){
        t.bookTicket();
    }
}
```

```
public class DelegationDemonstration {

    public static void main(String[] args) {
     // Here TicketBookingByAgent class is internally
     // delegating train ticket booking responsibility to other class
        TicketBookingByAgent agent = new TicketBookingByAgent(new TrainBooking());
        agent.bookTicket();

        agent = new TicketBookingByAgent(new AirBooking());
        agent.bookTicket();
    }
}
```

Applicability

- Use the *Delegation* in order to achieve the following
  - Reduce the coupling of methods to their class
  - Components that behave identically, but realize that this situation can change in the future.
  - If you need to use functionality in another class but you do not want to change that functionality then use delegation instead of inheritance.