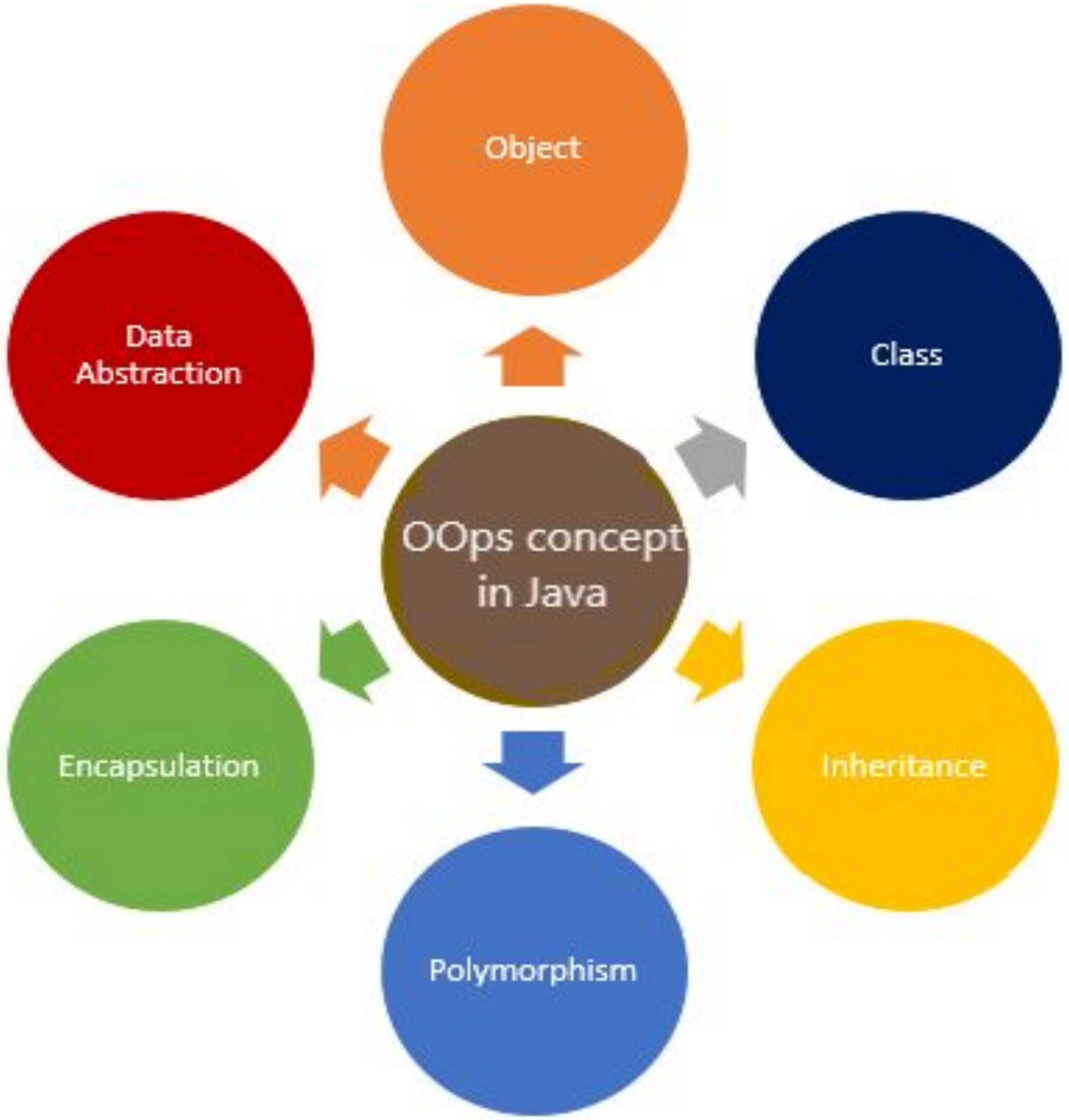


CS20004: Object Oriented Programming using Java

Lec-18

In this Discussion . . .

- Association
  - Aggregation
  - Composition
- Object class in Java
- References

Introduction

- Relationships between classes are crucial in object-oriented programming.
- Just as the concepts like classes and objects in object-oriented programming are built to model real-world entities, the relationships between classes in object-oriented programming are built to model the relationships between real-world entities, that these classes represent.

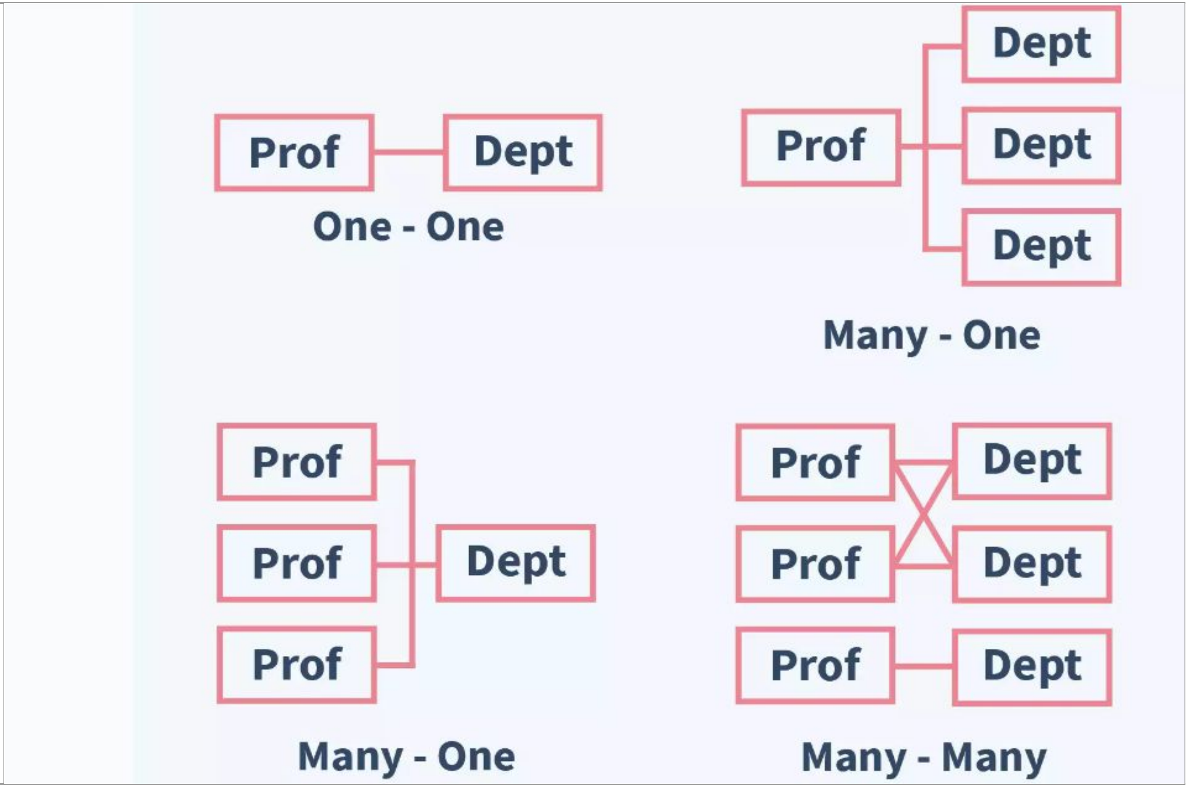
Association in Java

- Association, refers to the relationship between any two entities.
- **Association in java is the relationship that can be established between any two classes**. Such relationships can be expressed as the following four types:
    - One-to-One relation
    - One-to-many relation
    - Many-to-one relation
    - Many-to-many relation

Association in Java Example

- Let’s take help of two classes, Professor class, and Department class.
- Then the type of relationships/associations that can be possible between them, include:

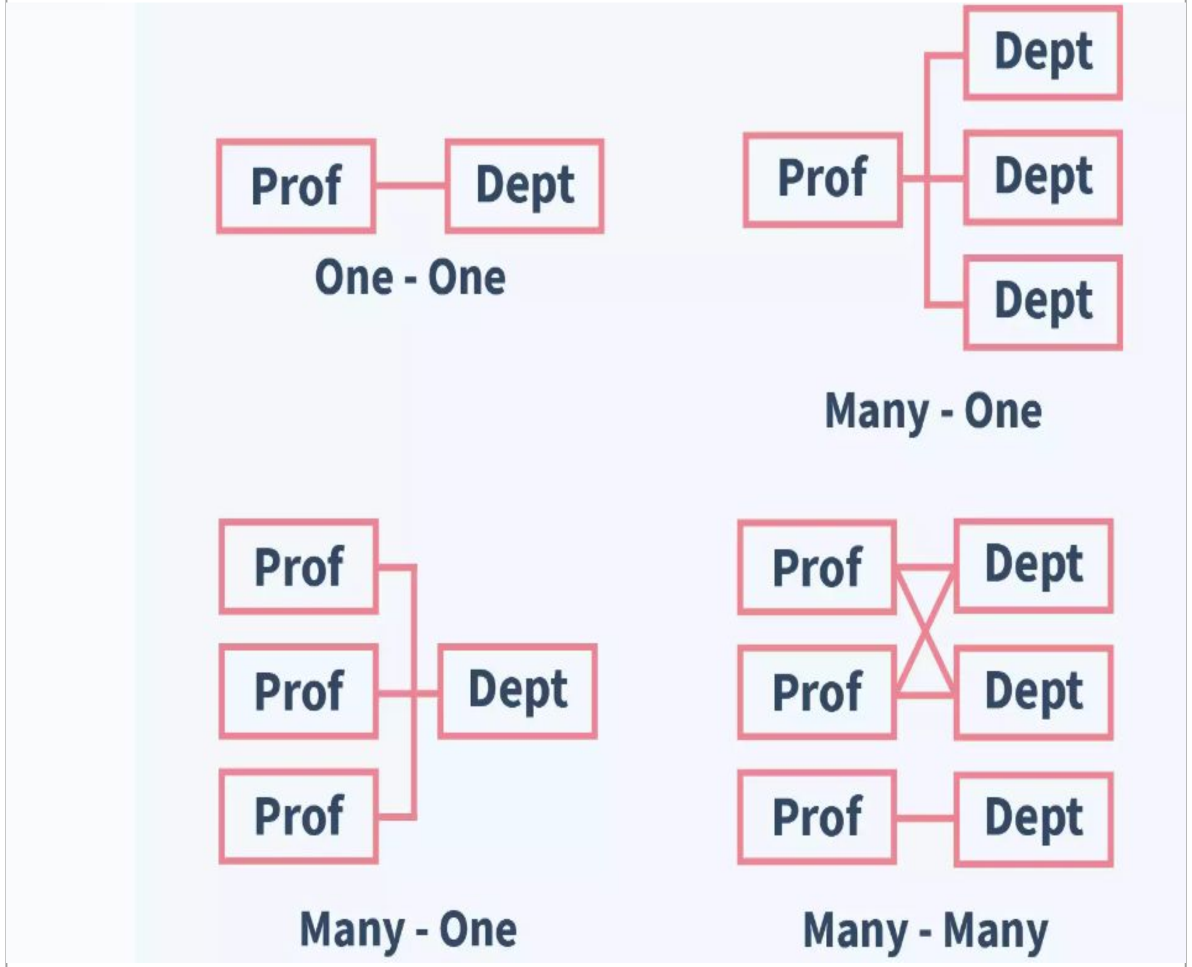
- One professor can only be assigned to work in one department. This forms a **one-to-one** association between the two classes.
- One professor can be assigned to work in multiple departments. This is a **one-to-many** association between the two classes.
- Multiple professors can be assigned to work in one department. This forms a **many-to-one** association between the two classes.
- Multiple professors can be assigned to work in multiple departments. This is the **many-to-many** association between the two classes.



Association in Java Example

- These associations in java help the objects of one class to communicate with the objects of the other class.

- **So one object will be able to access the data of another object.**
  - For example, A professor entity modeled as an object would be able to access/know the names of all the departments he works at. And a department object can be able to access/know the names/details of all the professors that work in it.
- **Functionality/Services of one object can be accessible to another object.**
  - For example, A professor who is trying to enroll in a department can be able to verify whether a department he wants to join has a vacancy. This service(programmatic method/function) to find whether there's a departmental vacancy can be provided by the Department class which the Professor class can access.



Association (one-to-many) in Java Example

https://docs.google.com/document/d/1j7g_vTDuoqaJjeV_Yzv3JUkxu69-ibeV3nvFPB0J--g/edit?usp=sharing

Forms of Object Relationships in Java

- There can be two types of object relationships in OOPs:

IS-A (Inheritance)	HAS-A (association)
• The **IS-A** relationship is nothing but Inheritance. The relationships that can be established between classes using the concept of inheritance are called IS-A relations. • Ex: A parrot is-a Bird. Here Bird is a base class, and Parrot is a derived class, Parrot class inherits all the properties and attributes & methods (other than those that are private) of base class Bird, thus establishing inheritance(IS-A) relation between the two classes.	• The **HAS-A** association is where the Instance variables of a class refer to objects of another class. In other words, one class stores the objects of another class as its instance variables thereby establishing a HAS-A association between the two classes. • The example program in previous slide **HAS-A** association. • As Department class is storing the objects of Professor class as its instance variable staff (the List Which is storing a list of Professors class objects). And a Department class object can access these stored Professor objects to store/retrieve information from Professor Class, thereby creating an association between the two classes.

Forms of Association

- Two forms of Association are possible in Java:
  - **Aggregation**
  - **Composition**

Forms of Association: Aggregation

- Aggregation in java is a form of HAS-A relationship between two classes.
  - It is a relatively more loosely coupled relation than composition in that, although both classes are associated with each other, one can exist without the other independently.
- Thus, **Aggregation in java** is **also called** a **weak association**.

Forms of Association: Aggregation Example

- Consider the association between a Country class and a Sportsperson class. Let’s define it as:
  - **Country** class is defined with a name and other attributes like size, population, capital, etc, and a list of all the Sportsperson(s) that come from it.
  - A **Sportsperson** class is defined with a name and other attributes like age, height, weight, etc.

Forms of Association: Aggregation Example (Contd.)

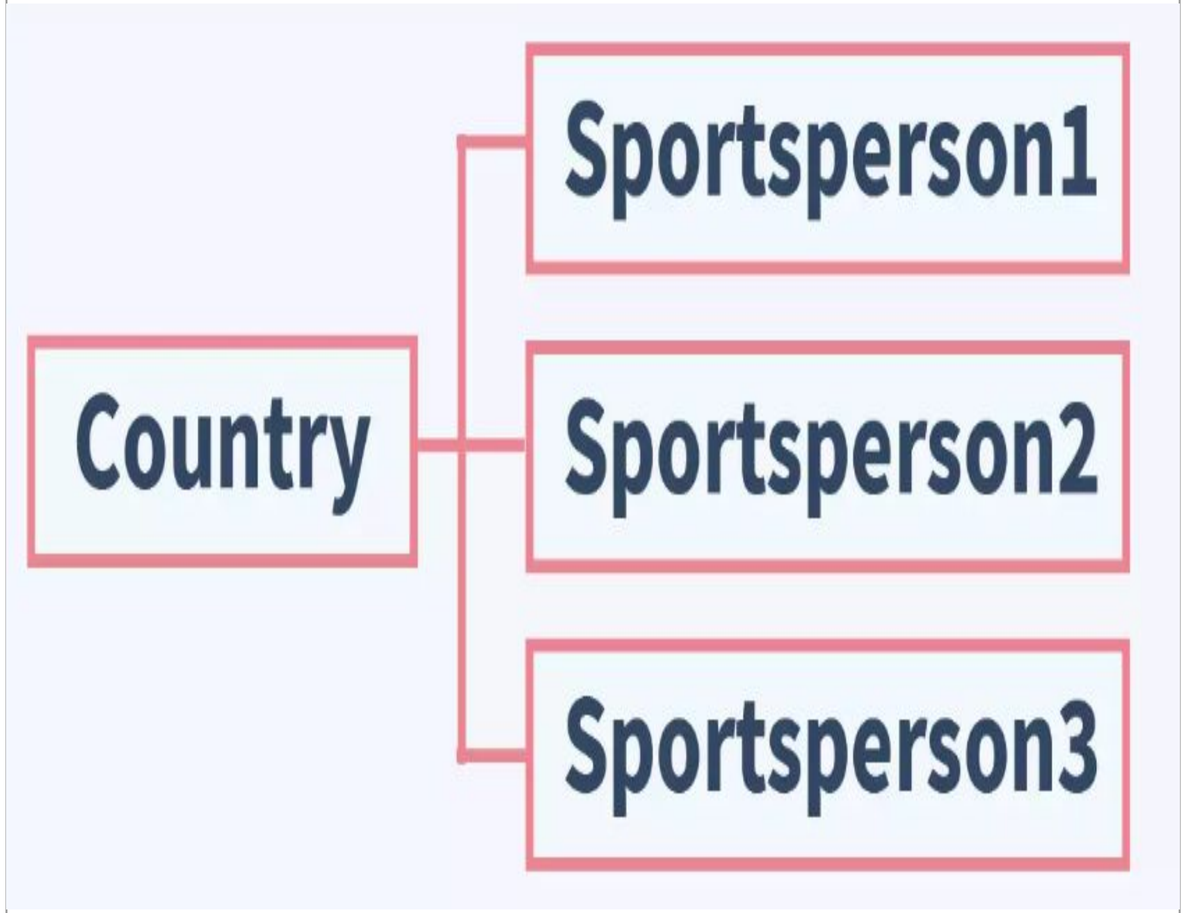
- Consider the association between a Country class and a Sportsperson class. Let’s define it as:
  - **Country** class is defined with a name and other attributes like size, population, capital, etc, and a list of all the Sportsperson(s) that come from it.
  - A **Sportsperson** class is defined with a name and other attributes like age, height, weight, etc.

Forms of Association: Aggregation Example (Contd.)

- In a real-world context, we can infer an association between a country and a sports person that hails from that country.
- Modeling this relation to OOPs, a Country object has-a list of Sportsperson objects that are related to it.
- Note that a sportsperson object can exist with his own attributes and methods, alone without the association with the country object.
- Similarly, a country object can exist independently without any association to a sportsperson object.
- In, other words both Country and Sportsperson classes are independent although there is an association between them.
- **Hence Aggregation is also known as a weak association.**

Forms of Association: Aggregation Example (Contd.)

- Another point to note is that here, the Country object has-a Sportsperson object and not the other way around,
  - meaning the Country object instance variables store the Sportsperson objects.
- So, the association is **one-sided**. **Thus, Aggregation is also known as a unidirectional association.**



Forms of Association: Composition

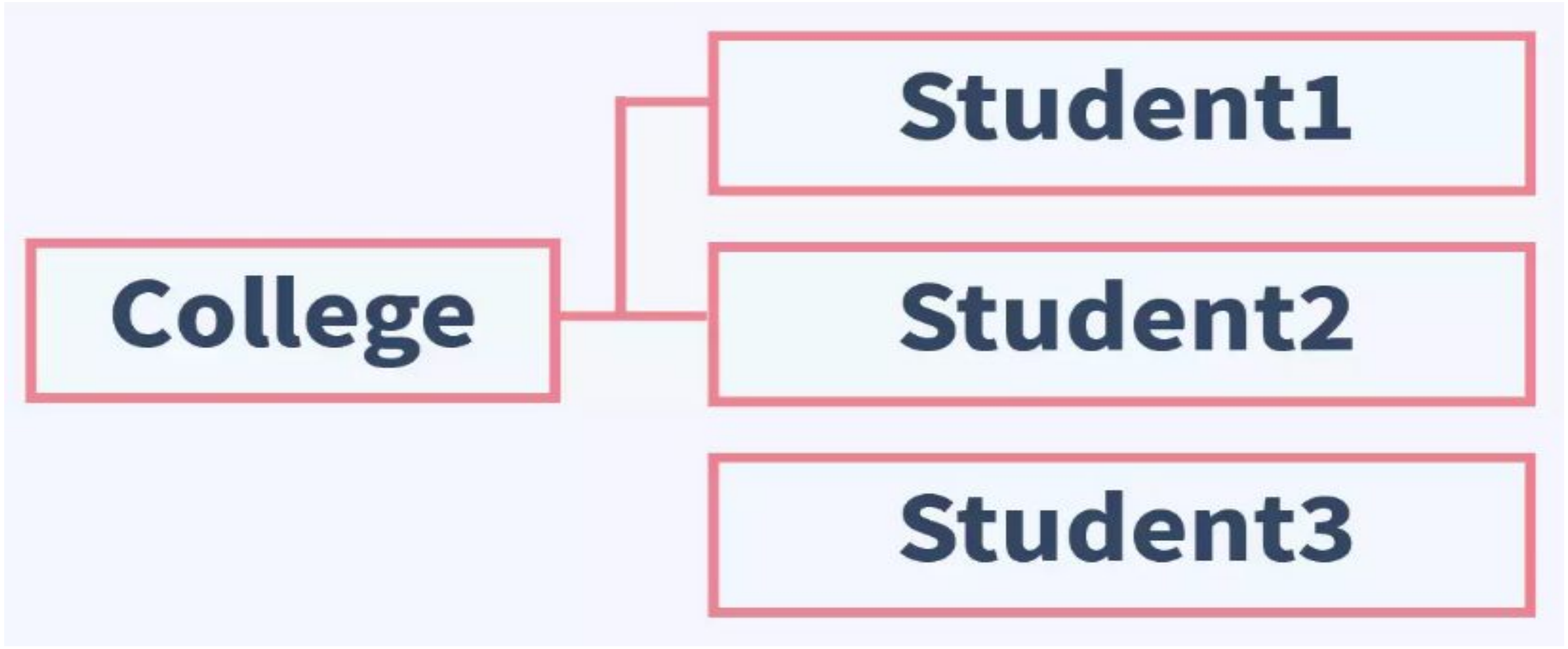
- Composition in java is a form of relation that is more **tightly coupled**. **Composition in java** is also **called Strong association**.
    - This association is also known as **Belongs-To** association as one class, for all intents and purpose belongs to another class, and exists because of it.
- In a Composition association, the classes cannot exist independent of each other.

- If the larger class which holds the objects of the smaller class is removed, it also means logically the smaller class cannot exist.

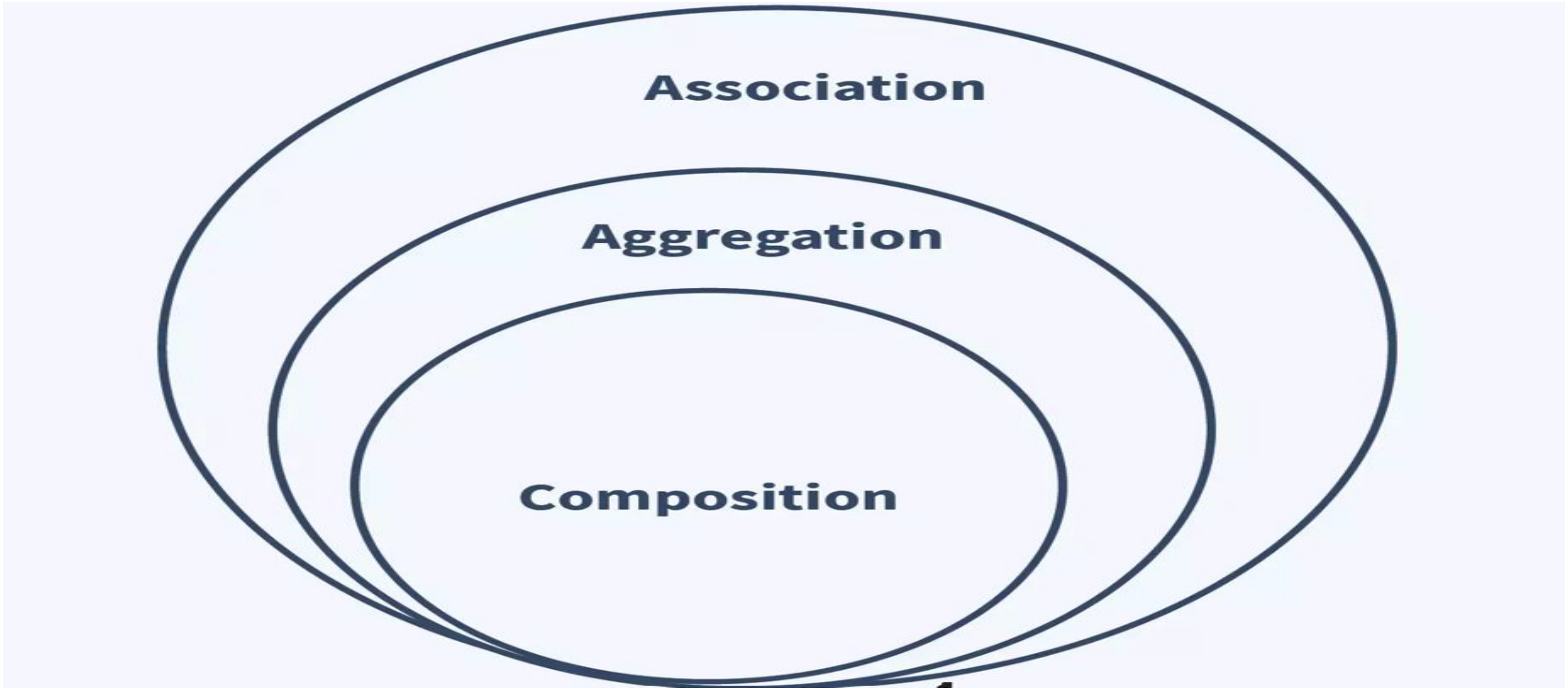
Forms of Association: Composition Example

- The association between College and Student. Let’s define it as:
    - **College** class is defined with name and the list of students that are studying in it
    - A **Student** class is defined with name and the college he is studying at.
- Here a student must be studying in at least one college if he is to be called Student. If the college class is removed, Student class cannot exist alone logically, because if a person is not studying in any college then he is not a student.

Forms of Association: Composition Example (Contd.)



Relationship between Association, Composition, and Aggregation



Aggregation Vs. Composition

Aggregation	**Composition**
● Weak association	● Strong association
● Classes in relation can exist independently	● One class is dependent on Another Independent class. The Dependent class cannot exist independently in the event of the non-existence of an independent class.
● One class has-a relationship with another class	● One class belongs-to another class
● Helps with code reusability. Since classes exist independently, associations can be reassigned or new associations created without any modifications to the existing class.	● Code is not that reusable as the association is dependent. Such Associations once established will create a dependency, and these associations cannot be reassigned or new associations like aggregation, etc cannot be created without changing the existing class.

Object Class in Java

- Object Class in Java is the topmost class among all the classes in Java.
    - Object class in Java is the parent class for all the classes.
        - It means that all the classes in Java are derived classes and their base class is the Object class.
- All the classes directly or indirectly inherit from the Object class in Java.

Object Class in Java

class Vehicle{
//body of class Vehicle
}
class Car extends Vehicle{
//body of class Car
}
	● Here we can see that the Car class directly inherits the Vehicle class. The **extends** keyword is used to establish inheritance between two classes.
	● We can see that the class Vehicle is not inheriting any class, **but it inherits the Object class. It is an example of the direct inheritance of object class**.
	● As Car inherits the Vehicle and the Vehicle inherits the Object class, **the Car indirectly inherits the Object class.**

Methods of Object Class

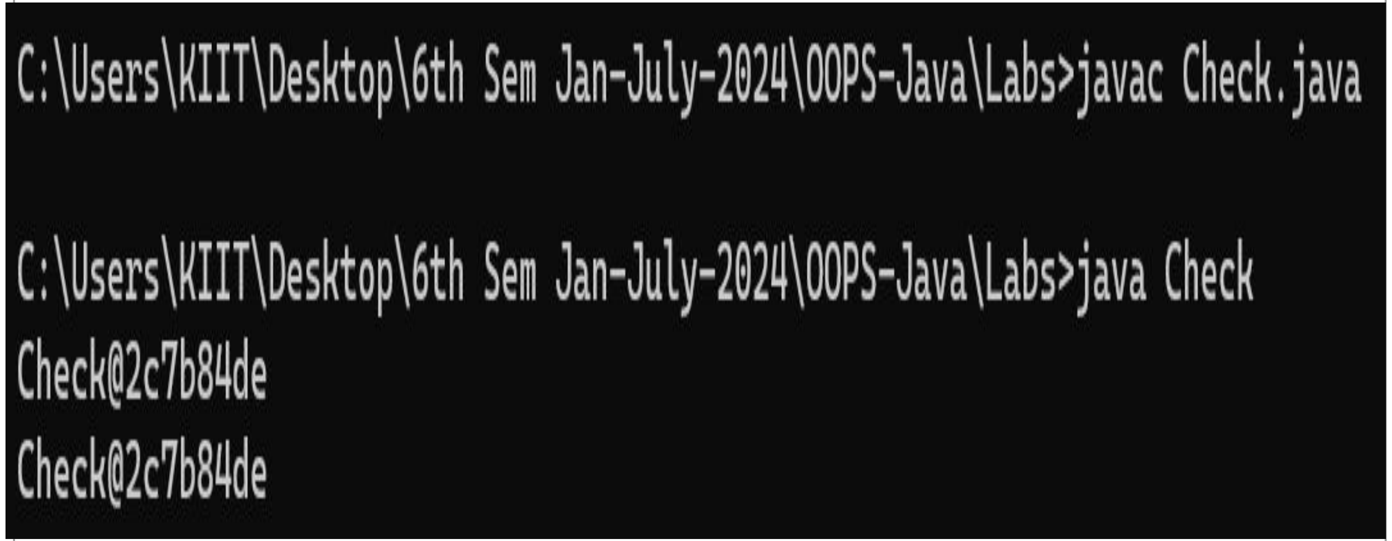
- There are various methods of the Object class. They are inherited by other classes.
  - **toString()** Method
    - It returns a string representation of an object. It is used to convert an Object "to String".
    - As each class is a subclass of the Object class in Java. So the toString() method gets inherited by the other classes from the Object class.
    - If we try to print the object of any class, then the JVM internally invokes the toString() method.

It is a good practice to override the toString() in an inherited class as it'll aid in a better string representation of the object. **If we do not override it,** it will give the ***hexadecimal representation*** of the object.

Methods of Object Class

- **toString()** Method:

```
import java.util.*;

public class Check{
    int car_no;

    Check(int car_no){
        this.car_no=car_no;
    }

    // Driver code
    public static void main(String args[])
    {
        Check s = new Check(16);

        // Below two statements are equivalent

        System.out.println(s.toString());
        System.out.println(s);
    }
}
```



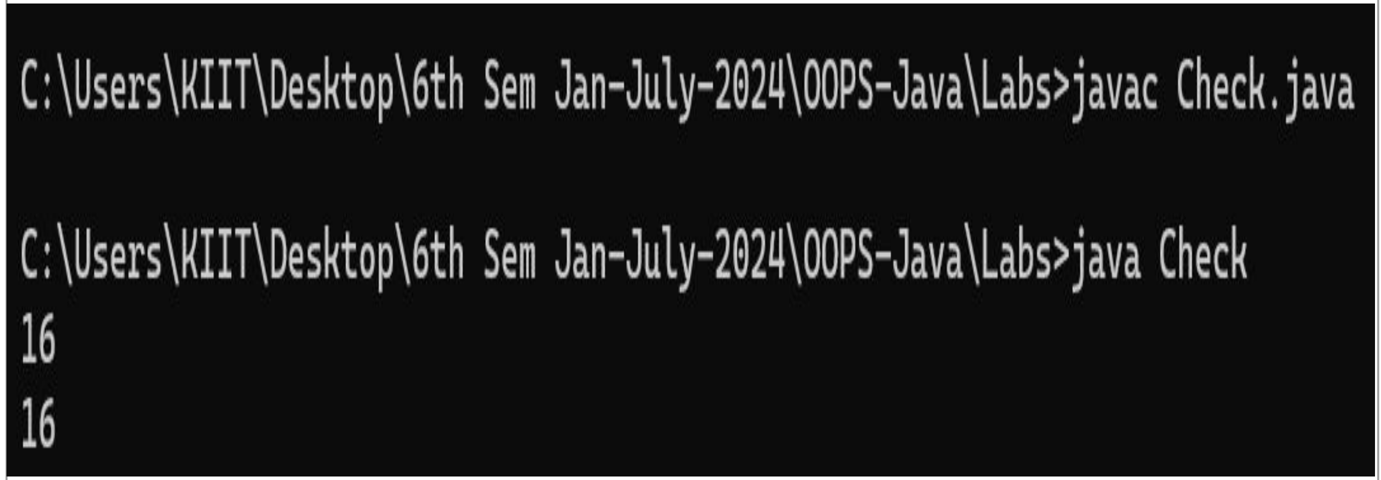
```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac Check.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java Check
Check@2c7b84de
Check@2c7b84de
```

Here we can see that when we tried to print the string representation of car_no using toString() method and print statement, it printed its hexadecimal representation. It is not the desired result.

Methods of Object Class

- **toString()** Method:

```
import java.util.*;
public class Check {
    int car_no;
    Check(int car_no){
        this.car_no=car_no;
    }
    // Overriding the toString()
   public String toString() {
        return car_no + " ";
    }
    // Driver code
    public static void main(String args[])
    {
        Check s = new Check(16);

        // Below two statements are equivalent

        System.out.println(s.toString());
        System.out.println(s);
    }
}
```



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac Check.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java Check
16
16
```

So, here we override the toString() method so that we get the exact value as a string that the variable has stored.

Methods of Object Class

- **hashCode()** Method
  - A hash code is an integer value that gets generated by the hashing algorithm.
  - Hash code is associated with each object in Java and is a distinct value.
  - It converts an object's internal address to an integer through an algorithm. It is not the memory address, it is the **integer representation of the memory address**.

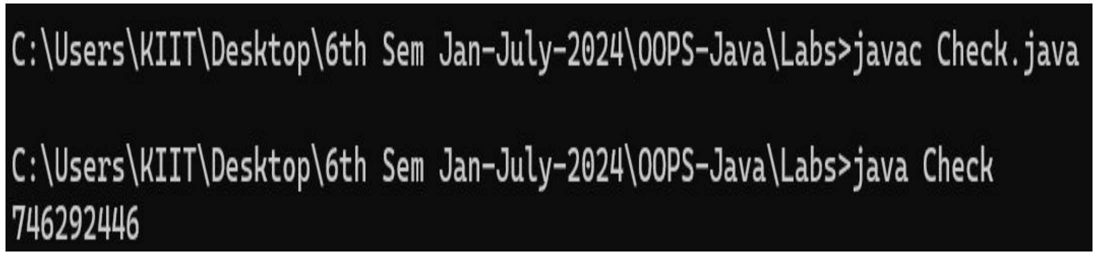
Syntax:

```
public int hashCode()
```

Methods of Object Class

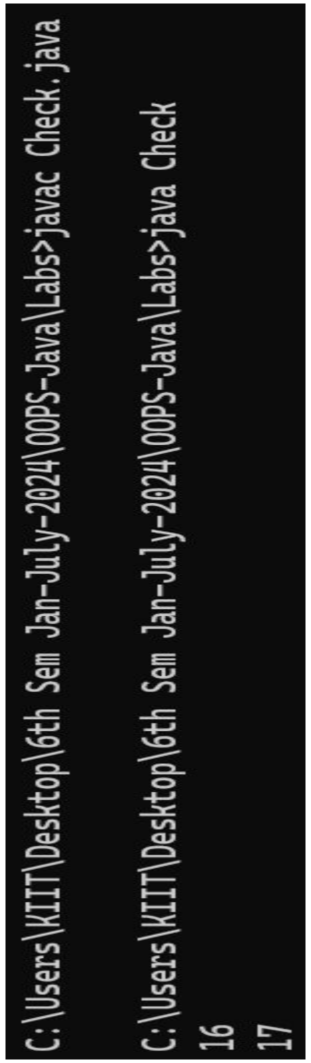
- **hashCode()** Method

```
import java.util.*;

public class Check {
    int car_no;

    Check(int car_no){
        this.car_no=car_no;
    }

    // Driver code
    public static void main(String args[])
    {
        Check s = new Check(16);

        System.out.println(s.hashCode());
    }
  }
```



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac Check.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java Check
746292446
```

```
import java.util.*;

public class Check {
    int car_no;

    Check(int car_no){
        this.car_no=car_no;
    }

  //Overriding the hashCode()
    @Override public int hashCode() {
        return car_no;
    }

    // Driver code
    public static void main(String args[])
    {
        Check s = new Check(16);
        Check ss = new Check(17);

        System.out.println(s.hashCode());
        System.out.println(ss.hashCode());
    }
  }
```



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac Check.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java Check
16
17
```

Methods of Object Class

- **equals (Object obj)** Method
  - This method compares two objects and returns whether they are equal or not.
  - It is used to compare the value of the object on which the method is called and the object value which is passed as the parameter.
  - It is usually recommended to override the hashCode() method whenever this method is overridden, as it'll ensure that equal objects must have equal hash codes.

```
public boolean equals(Object obj)
{
        return (this == obj);
}
```
	• The default implementation checks whether the two objects i.e., the object on which the method is called and the obj object are equal or not.
	• Here equal means whether they are referring to the same object. If they are the same, it returns true, else it returns false.

Methods of Object Class

- **getClass()** Method
  - It is used to return the class object of this object.
  - Also, it fetches the actual runtime class of the object on which the method is called.
  - This is also a native method.
  - It can be used to get the metadata of the this class. Metadata of a class includes the class name, fields name, methods, constructor, etc.

Methods of Object Class

- **clone()** Method
  - The clone() method is used to create an exact copy of this object. It creates a new object and copies all the data of the this object to the new object.
- **notify()** Method
  - The notify() method is used to wake up only one single thread that is waiting on the object, and that thread starts the execution.
  - If any threads are waiting on this object, any one of them is arbitrarily chosen to be awakened.
  - This method does not return any value.

Methods of Object Class

- **notifyAll()** Method
  - The notifyAll() method is used to wake up all threads that are waiting on this object. This method gives the notification to all waiting threads of an object.
- **wait()** Method
  - The wait() method tells the current thread to give up the lock and go to sleep. It happens until some different thread enters the same monitor and calls the methods like -notify() or notifyAll() method.
- **wait(long timeout)** Method
  - The wait(long timeout) method makes the current thread wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

Methods of Object Class

- **wait(long timeout, int nanos)** Method
    - This is another modification of the wait() method. Here two values are passed as parameters.
        - **timeout** – the maximum time to wait in milliseconds.
        - **nanos** – additional time, in nanoseconds range 0-999999.

References

1. https://www.scaler.com/topics/association-composition-and-aggregation-in-java/
2. https://www.scaler.com/topics/object-class-in-java/
3.