

Lab 07

Class Relationships in Java | Types of Relationships

Class relationships in Java define the special relationships among different kinds of classes.

For example, there is a special relationship between a class named Vehicle and a class Car: A Car is a type of Vehicle.

There is a different kind of relationship among classes Shape, Circle, Rectangle, and Square. A Circle is a type of Shape. A Rectangle is a type of Shape.

When we design a major application or program, we need to explore the relationships among classes. It helps us in a number of ways.

For example, suppose in an application, we have classes with common behaviors (methods) then we can save effort by placing the common behaviors (methods) within the superclass.

Suppose some classes are not related to each other, then we can assign different programmers to implement each of them, without worrying that one of them will have to wait for the other.

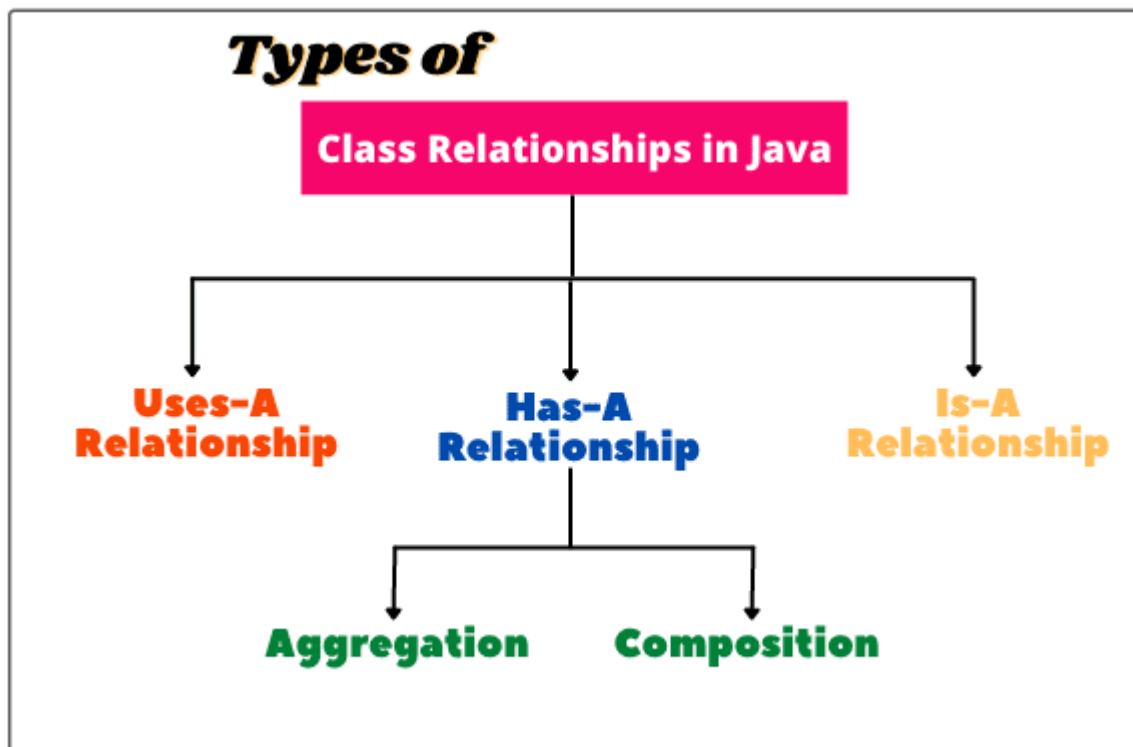
For this purpose, we need to learn different types of relationships among classes in Java. Relationships among classes help to understand how objects in a program work together and communicate with each other.

Types of Relationship among Classes in Java

There are three most common relationships among classes in Java that are as follows:

Lab 07

- a. Dependence ("Uses-A")
- b. Association ("Has-A")
- c. Inheritance ("Is-A")



Association is further classified into aggregation and composition that will be understood in the further tutorial in detail.

Let's understand these three relationships among classes one by one.

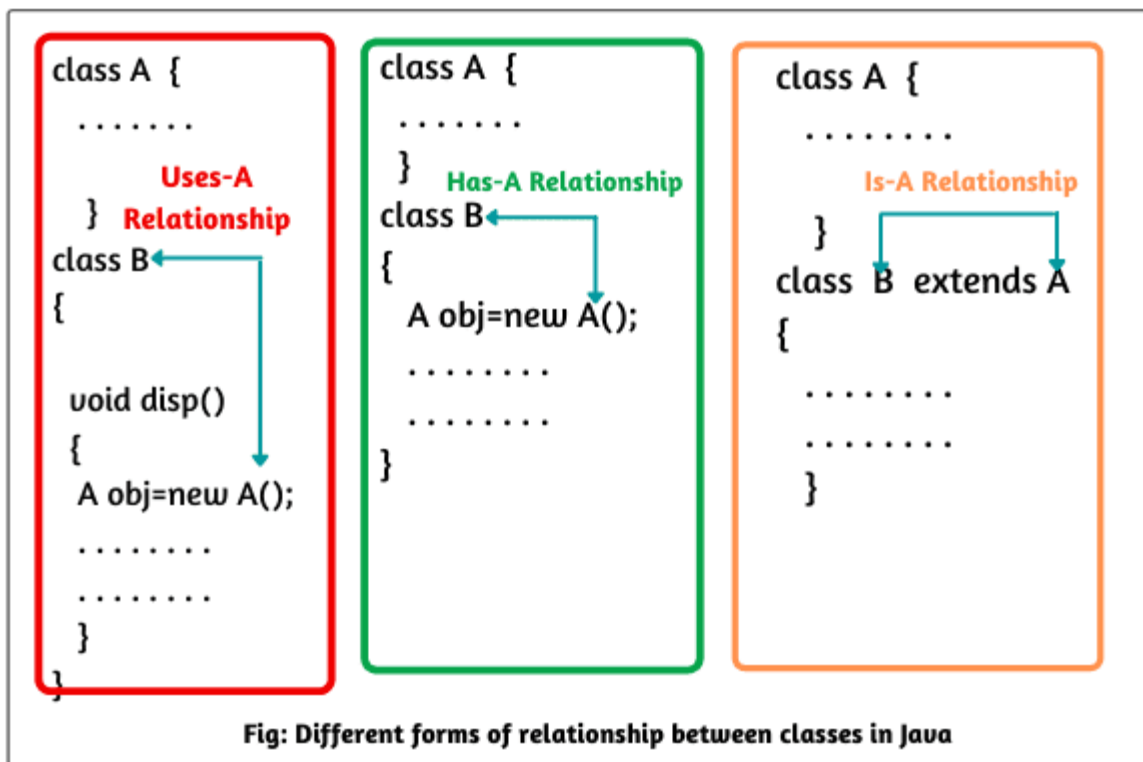
Dependence (Uses-A) Relationship in Java

When we create an object of a class inside a method of another class, this relationship is called **dependence relationship in Java**, or simply Uses-A relationship.

Lab 07

In other words, when a method of a class uses an object of another class, it is called dependency in java. It is the most obvious and most general relationship in java.

Look at the below figure where a method display() of class B uses an object of class A. So, we can say that class A depends on another class B if it uses an object of class A.



If several classes of an application program depend on each other, then we say that the coupling between classes is high.

It is a good programming practice to minimize the dependency between classes (i.e., coupling) because too many dependencies make an application program difficult to manage.

On the other hand, if there are few dependencies between classes, then we say that the coupling between classes is low.

Lab 07

Why does coupling matter?

If a class changes its behavior in the next release of the application program, all the classes that depend on it may also be affected. In this situation, we will need to update all the coupled classes.

Therefore, if the coupling between classes will be low, we can easily manage them. Thus, we must remove unnecessary coupling between classes.

Association (“Has-A”) Relationship in Java

Association is another fundamental relationship between classes that is informally known as “**Has-A**” relationship.

When an object of one class is created as data member inside another class, it is called **association relationship in java** or simply Has-A relationship.

Look at the above figure where an object of class A is created as data member inside another class B. This relationship is known as Has-A relationship. It is easy to understand and makes a stronger form of dependency.

Inheritance (“Is-A”) Relationship in Java

Inheritance represents Is-a relationship in Java. It establishes a relationship between a more general class (known as superclass) and a more specialized class (known as subclass).







Lab 07

In other words, Is-A relationship defines the relationship between two classes in which one class extends another class.

Look at the above figure where a class B makes a relationship with class A by the keyword “extends” and can inherit data members from class A.

UML Notation in Java

UML stands for Unified Modeling Language. It is an international standard notation. Many Programmers use this notation to draw classes diagram that explains the relationship between classes.

UML NOTATION FOR CLASS RELATIONSHIPS	
Relationship	UML Connector
Inheritance	
Interface Inheritance	
Dependency	
Aggregation	
Association	
Direct Association	

There is a number of tools available for drawing UML diagrams. A table in the below figure shows the UML notation for class relationships in Java.

Lab 07

Has-A Relationship in Java | Types, Example

Has-A Relationship in Java | In addition to Uses-A relationship and Is-A relationship that represents **inheritance**, one more relationship is commonly used in **object-oriented programming**. This relationship is called Has-relationship in Java.

When an object of one class is created as a data member inside another class, it is called Has-A relationship.

In other words, a relationship in which an object of one class has a reference to an object of another class or another instance of the same class is called Has-A relationship in Java.

Let's understand Has-A relationship with the help of different kinds of examples.

Has-A Relationship Examples

1. A most common example of Has-A relationship in Java is "A person has an address".

Has-A relationship denotes a **whole-part relationship** where a part cannot exist without the whole. In the above example, the person represents the whole and the address represents the part. A part cannot exist by itself.

In Java, there is no special feature by which we can identify the Has-A relationship in the code.

Lab 07

Person class has an instance variable of type Address as shown in the below code. An instance of Address class is created outside of Person class.

```
public class Address {  
  
    // Code goes here.  
  
}  
  
public class Person {  
  
    // Person has-a Address.  
  
    Address addr = new Address();  
  
    // Other codes go here.  
  
}
```

Look at the above figure where an object of class Address is created as a data member inside another class Person. This relationship is known as Has-A relationship.

Lab 07

```
class Address
{
    .....
}
class Person
{
    Address addr = new Address();
    .....
    .....
}
```




Fig: Has-A Relationship in Java

In Java, there is no such keyword that implements a Has-A relationship. But we mostly use new keywords to implement a Has-A relationship in Java.

Let's take another common example to understand Java Has-A relationship.

2. We know that a CPU is part of the computer. We can also rephrase this relationship as "A computer has a CPU".

Does the existence of CPU outside a computer make any sense?

The answer is no. The existence of a CPU makes sense only within the computer. So, a computer and CPU represent the whole-part relationship.

Look at the below code to understand better.

Lab 07

```
public class CPU {  
  
    // Code goes here.  
  
}  
  
public class Computer {  
  
    // CPU part-of Computer.  
  
    private CPU cpu = new CPU();  
  
    // Other codes go here.  
  
}
```

3. We know that a brain has a thought. Thought cannot exist without the existence of the brain. The code is given below:

```
public class Thought {  
  
    // Code goes here.  
  
}  
  
public class Brain {  
  
    Thought thought = new Thought();  
  
    // Other codes go here.
```

Lab 07

4. We know about a special class known as inner class in Java that can also be used to represent Has-A relationship. An instance of inner class can exist only inside an instance of its outer class (enclosing class). The outer class would be the whole and the inner class would be the part.

How to decide which type of Relationship we need?

We know that relationships between objects make all the differences in object-oriented programming. The most important relationships are Is-A relationship and Has-A relationship.

The best way to decide which kind of relationship we need is as follows:

- a. If your problem with a phrase contains "... is a ...". Then you should use Is-A relationship (Inheritance). For example, "A dog is a pet". We cannot say "A dog has a pet", as makes no sense at all. So, in this case, we will create a superclass named Pet and a derived subclass named Dog.
- b. On the other hand, if your problem with a phrase like this: "A pet has a name" then use has-a relationship. This is because if we use Is-A relationship instead of Has-A relationship, then the statement will be like this: "A dog is a name". This statement makes no sense at all.

From all the above examples, you will have understood which kind of relationship you need in your code.

Types of Has-A Relationship in Java

Lab 07

There are two types of Has-A relationship that are as follows:

- a. Aggregation
- b. Composition

Aggregation: Aggregation is one of the core concepts of object-oriented programming. It focuses on establishing a Has-A relationship between two classes.

In other words, two aggregated objects have their own life cycle but one of the objects has an owner of Has-A relationship and child object cannot belong to another parent object.

For example, a library has students. If the library is destroyed, students will exist without library.

Composition: Composition is another one of the core concepts of object-oriented programming. It focuses on establishing a strong Has-A relationship between the two classes.

In other words, two composited objects cannot have their own life cycle. That is, a composite object cannot exist on its own. If one composite object is destroyed, all its parts are also be deleted.

For example, a house can have multiple rooms. A room cannot exist independently and any room cannot belong to two different houses. If the house is destroyed, all its rooms will be automatically destroyed.

Types of Association in Java

Lab 07

There are two combinations of association in java that are as follows:

a. Aggregation:

An aggregation is a special form of unidirectional association that represents an ownership relationship between two objects.

That is, two aggregated objects have their own life cycles but one of the objects is the owner of the Has-A relationship. It is often referred to as a part-of relationship.

For example, the association between a car and the passengers in the car is aggregation. Passengers can be the owner of car. Both have their own life cycle but one of the passengers can be the owner of car. Here, car is a part-of relationship.

You will learn in more detail about aggregation with example programs in the next tutorial.

b. Composition:

A composition is a special and more restrictive form of aggregation. It also represents Has-A relationship where an object cannot exist on its own. The "whole" is actually dependent on the "part".

For example, the association between a car and its engine represents composition. Since a car cannot exist without an engine.

We will know in more detail about composition in the further tutorial with realtime example programs.

Lab 07

Aggregation in Java OOPs | Example Program

Aggregation in Java is one of the core concepts of **object-oriented programming**. It focuses on establishing **Has-A relationship** between two classes.

Aggregation is a more specialized form of unidirectional **association** that represents an ownership relationship between two class objects.

In other words, when two aggregated objects have their own life cycle (i.e. independent lifetime) but one of the objects is the owner of Has-A relationship, it is called aggregation in java.

The owner object is called aggregating object and its class is called aggregating class. The aggregating class has a reference to another class and is the owner of that class.

Having their own relationship means that destroying one object will not affect another object. For example, a library has students. If the library is destroyed, students will exist without library.

Realtime Example of Aggregation Relationship in Java

A most common example of aggregating relationship is "A student has an address". A student has many pieces of information such as name, roll no, email id, etc.

It also contains one more important object named "address" that contains information such as city, state, country, zip code.

Let's implement this common example programmatically to understand the aggregation relationship in java.

Lab 07

In this example program, Student class has an object of Address class where the address object contains its own information such as city, state, country, etc. This relationship is Student Has-A address and is called aggregation.

Look at the program source code to understand better.

Program source code 1:

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

A student Has-A an address. Therefore, the Student class must be able to receive address information as follows:

```
public class Student {
```



Lab 07

```
String name;

int rollNo;

Address address;

int pinCode;

public Student(String name, int rollNo, Address address) {

    this.rollNo = rollNo;

    this.name = name;

    this.address=address;

}

void display(){

    System.out.println("Name: " +name + ", "+"Roll no: " +rollNo);

    System.out.println("Address:");

    System.out.println(address.city+" "+address.state+" "+address.country+ " "
+address.pinCode);

    System.out.println("\n");

}

public static void main(String[] args)

{
```



Lab 07

```
Address addr1 = new Address("Dhanbad","Jharkhand","India", 826001);
```

```
Address addr2 = new Address("Ranchi","Jharkhand","India", 825001);
```

```
Student st1 = new Student("Deep", 05, addr1);
```

```
Student st2 = new Student("John", 02, addr2);
```

```
st1.display();
```

```
st2.display();
```

```
}
```

```
}
```

Output:

Name: Deep, Roll no: 5

Address:

Dhanbad, Jharkhand, India, 826001

Name: John, Roll no: 2

Address:

Ranchi, Jharkhand, India, 825001

Lab 07

The complete application program is named Aggregation. As you can notice in the above program, a class Student contains a reference to class Address whose instance exists and are accessible outside of Student, we can say that Student is an aggregation of Address.

If Student is an aggregation of Address, we can say that a Student object "Has-A" Address object.

Moving on, let's take one more example program for practice based on it.

A football player has a football. It is a unidirectional relationship because a football can not have a football player. Even if football player dies, football will not be affected. So, let's write code for it.

Program source code 2:

```
public class Football {  
  
    private String type, size, weight;  
  
    Football(String type, String size, String weight){  
  
        this.type = type;  
  
        this.size = size;  
  
        this.weight = weight;  
  
    }  
  
    public String getType(){  
  
        return type;  
  
    }  
}
```



Lab 07

```
}

public void setType(String type){

    this.type = type;

}

public String getSize(){

    return size;

}

public void setSize(String size){

    this.size = size;

}

public String getWeight(){

    return weight;

}

public void setWeight(String weight){

    this.weight = weight;

}

}

public class FootballPlayer {
```



Lab 07

```
private String name;

private Football football;

FootballPlayer(String name, Football football){

    this.name = name;

    this.football = football;

}

public String getName(){

    return name;

}

public void setName(String name){

    this.name = name;

}

public Football getFootball(){

    return football;

}

public void setFootball(Football football){

    this.football = football;
```

Lab 07

```
}  
  
public static void main(String[] args)  
{  
  
    Football football = new Football("Association Football", "68-70 cm", "420 gm" );  
  
    FootballPlayer fbp = new FootballPlayer("John", football);  
  
    System.out.println("Player " +fbp.getName() + " plays with "  
+fbp.getFootball().getType());  
  
    }  
}
```

Output:

Player John plays with Association Football

When to use Aggregation in Java?

Aggregation is used for code reusability if there is no Is-A relationship. Let's see a simple example program based on it.

Program source code 3:

```
public class Calculation  
{  
  
    int calArea(int length, int breadth){  
  
        return (length * breadth);  
    }  
}
```



Lab 07

```
}  
  
}  
  
public class Rectangle {  
  
    Calculation cal; // Use of Aggregation.  
  
  
    int area(int length, int breadth){  
  
        cal = new Calculation();  
  
        int areaRec = cal.calArea(length, breadth); // code reusability.  
  
        return areaRec;  
  
    }  
  
}  
  
public class Test {  
  
    public static void main(String[] args)  
  
    {  
  
        Rectangle rec = new Rectangle();  
  
        int result = rec.area(25, 60);  
  
        System.out.println("Area of rectangle: " +result);  
  
    }  
  
}
```

Lab 07

```
}
```

```
}
```

Output:

Area of rectangle: 1500

Difference between Association vs Aggregation

Both association and aggregation are the core concepts of OOP and represent Has-A relationship in Java. But there are also certain important differences between association and aggregation that are as follows:

1. Association establishes the relationship between two classes that are independent of each other whereas, aggregation establishes an ownership relationship between two classes.
2. Association has no owner whereas, aggregation has ownership of association.
3. Association can have relationship such as unidirectional/bidirectional, one-to-one, one-to-many, many-to-one, and many-to-many whereas, aggregation has a unidirectional relationship.

Composition in Java | Example Program

Lab 07

Composition in Java is one of the core concepts of object-oriented programming. It is different from **inheritance**.

Inheritance is defined as Is-A relationship whereas, composition is defined as Has-A relationship.

Both composition and inheritance are design techniques and can be used for the purpose of code reusability.

Composition is a more specialized form of **aggregation**. In other words, it is a more restrictive and strong form of aggregation.

Java composition differs from aggregation is that aggregation represents a Has-A relationship between two objects having their own lifetime, but composition represents a Has-A relationship that contains an object that cannot exist on its own.

In other words, child object does not have its own lifetime. If the parent object is destroyed, the child object will also be destroyed. The child object cannot exist without the existence of its parent object.

In simple words, a composition can be defined as if a parent object contains a child object and the child object cannot exist on its own without having the existence of parent object, it is called composition in java.

It can be achieved by using an instance variable that refers to another object.

Let's understand the basic meaning of java composition with help of realtime examples.

Realtime Example of Composition in Java

Lab 07

1. We live in a house. House can have many rooms. But there is no independent life of room and it cannot also associate to two different houses.

If we destroy the house, room will be automatically destroyed. So, we can say that a room is PART-OF the house.

2. Another realtime example of java composition is "Car and Engine". A car has an engine. In other words, an engine is a PART-OF car. Here, a car is a whole, and engine is a part of that car.

If the car is destroyed then its engine will be destroyed as well. So, without the existence of car, there is no life of an engine. The life of an engine is totally dependent on the life cycle of car.

Let's understand it programmatically with the help of an example program. Look at the following source code to understand better.

Program source code 1:

```
public class Engine {  
  
    private String type;  
  
    private int horsepower;  
  
    Engine(String type, int horsepower){  
  
        this.type = type;  
  
        this.horsePower = horsepower;  
  
    }  
  
    public String getType(){
```




Lab 07

```
return type;
}

public int getHorsePower(){
    return horsePower;
}

public void setType(String type){
    this.type = type;
}

public void setHorsePower(int horsePower){
    this.horsePower = horsePower;
}
}

public class Car
{
    private final String name;

    private final Engine engine; // Composition.

    public Car(String name, Engine engine){
```



Lab 07

```
this.name = name;

this.engine = engine;
}

public String getName(){

    return name;

}

public Engine getEngine(){

    return engine;

}

}

public class Test {

    public static void main(String[] args)

    {

        // Creating an object of Engine class.

        Engine engn = new Engine("Petrol", 300);

        // Creating an object of Car class.

        Car car = new Car("Alto", engn);
```

Lab 07

```
System.out.println("Name of car: " + car.getName() + "\n" + "Type of engine: "
+ engn.getType() + "\n" + "Horse power of Engine: " + engn.getHorsePower());

}

}
```

Output:

Name of car: Alto

Type of engine: Petrol

Horse power of Engine: 300

Features of Composition in Java

There are several important features about java composition that are as follows:

1. Composition represents a has-a relationship in java. In other words, it represents PART-OF relationship.
2. It is a more restrictive form of aggregation.
3. In composition, both the entities are associated with each other.
4. A composition between two entities happens when an object (in other words, parent object) contains composed object (in other words, child object), and the composed object cannot exist without the existence of that object.

For example, a library contains a number of books on the same or different subjects. If the library gets destroyed for any reason, all books placed within that library will be destroyed automatically. That is, books cannot exist without a library of school or college.



Lab 07

5. It can be achieved by using an instance variable that refers to other objects.

Advantages of using Composition over Inheritance

There are several benefits of using composition in java over inheritance. They are as follows:

1. Composition is better than inheritance because it allows to reuse of code and provides visibility control on objects.
2. Java doesn't allow multiple inheritance but by using composition we can achieve it easily.
3. Composition grants better test-ability of a class.
4. By using composition, we can easily replace the implementation of a composed class with a better and improved version.
5. Composition allows the changing of member objects at run time so that we can dynamically change the behavior of our program.