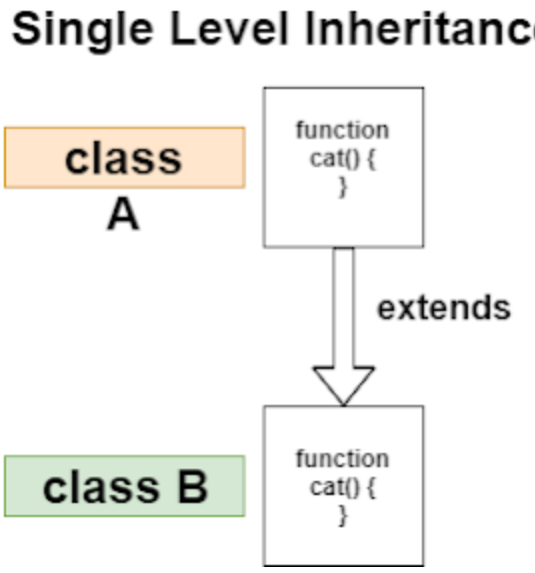
Inheritance in Java

It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition.

Why Do We Need Java Inheritance?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overloading: Method Overriding** is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. **Abstraction** only shows the functionality to the user.



```
class A{
}


class B extends A{
}
```

Parent Class: The class that shares the fields and methods with the child class is known as parent class, super class or Base class. In the above code, Class A is the parent class.

Child Class: The class that extends the features of another class is known as child class, sub class or derived class. In the above code, class B is the child class.

Terminologies used in Inheritance:

- Super class and base class are synonyms of Parent class.
- Sub class and derived class are synonyms of Child class.
- Properties and fields are synonyms of Data members.
- Functionality is a synonym of method.

Extends: In Java, the *extends* keyword is used to indicate that the **class** which is being defined is derived from the base class using inheritance.

So basically, extends keyword is used to extend the functionality of the parent class to the subclass.

```
package Inheritance;

public class Teacher {

    // fields of parent class
    String designation = "Teacher";
```

```
    String collegeName = "VSICS";

    // method of parent class
    void profession() {
        System.out.println("Teaching");
    }

}
```

```
package Inheritance;

public class ComputerScTeacher extends Teacher {

    String mainSubject = "CompSc";

    public static void main(String args[]) {
        ComputerScTeacher obj = new ComputerScTeacher();
        // accessing the fields of parent class
        System.out.println(obj.collegeName);
        System.out.println(obj.designation);

        System.out.println(obj.mainSubject);

        // accessing the method of parent class
        obj.profession();


    }
}
```

Based on the above example we can say that `CompscTeacher` **IS-A** `Teacher`. This means that a child class has **IS-A** relationship with the parent class. This is why

inheritance is known as **IS-A relationship** between child and parent class

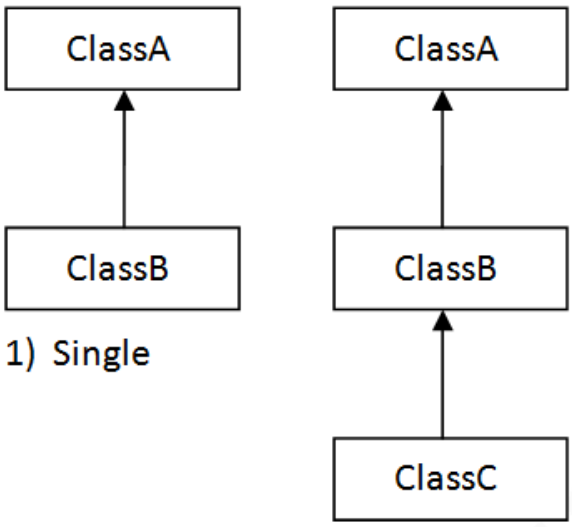
IS-A is a way of saying: **This object is a type of that object**.

```
public class Animal {
}
public class Mammal extends Animal {
}
public class Reptile extends Animal {
}
public class Dog extends Mammal {
}
```

Now, if we consider the IS-A relationship

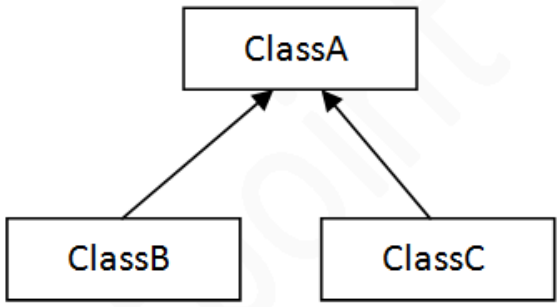
- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

Types of inheritance



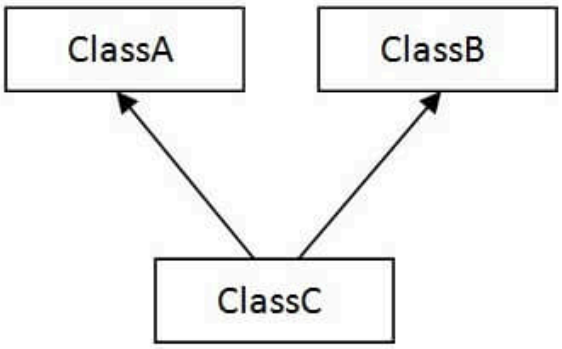
1) Single

2) Multilevel

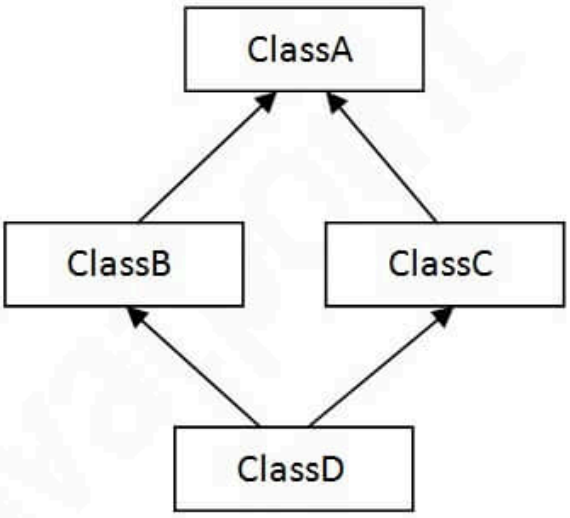


3) Hierarchical

Implement By Interface



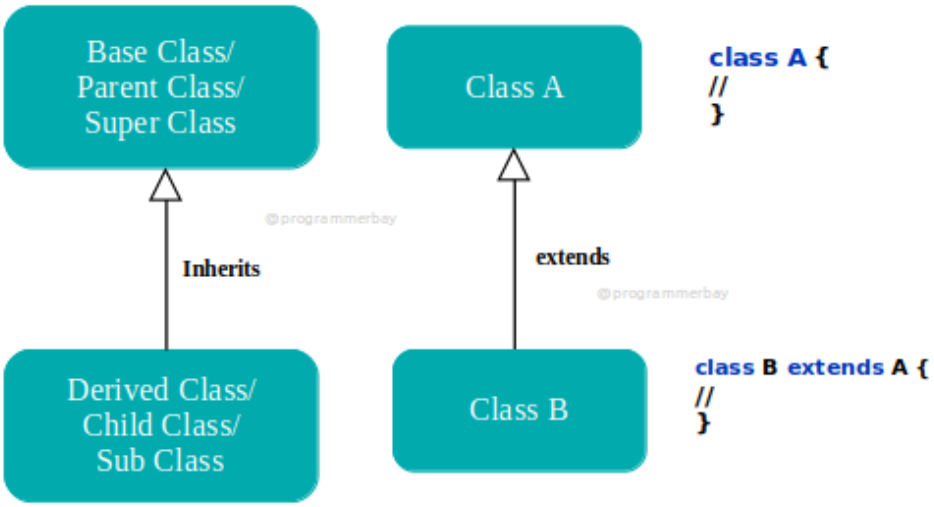
4) Multiple



5) Hybrid

.

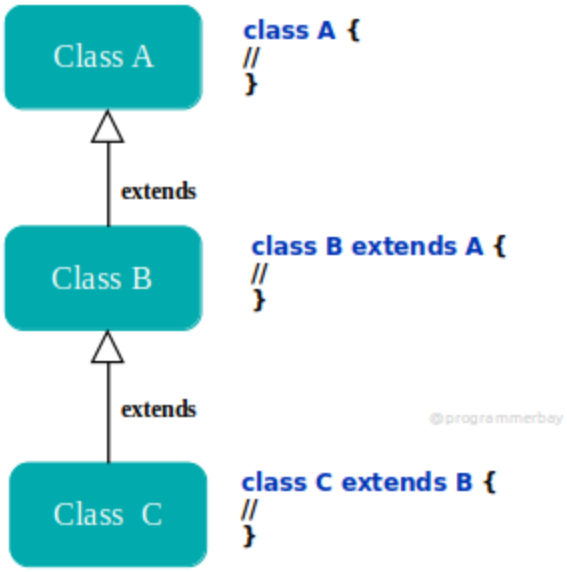
Single inheritance can be defined as a type of inheritance, where a single parent class is inherited by only a single child class.




```
public class A {
    public void display() {
        System.out.println("I am a method from class A");
    }
}

// B is inheriting display method of A
class B extends A {
    public void print() {
        System.out.println("I am a method from class B");
    }

    public static void main(String[] args) {
        B objB = new B();
        objB.display(); // Reusing the method of A named display
        objB.print();
    }
}
```

Multilevel inheritance is a type of inheritance where a subclass acts as a superclass of another class. In

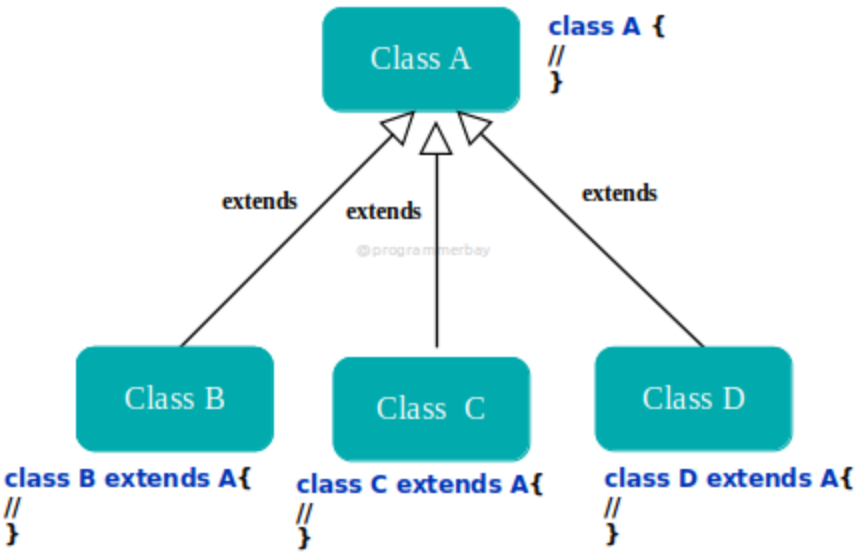
other words, when a class having a parent class, is extended by another class and forms a sequential chain, then it's termed Multilevel inheritance.




```
class A {
public void display() {
      System.out.println("I am a method from class A");
   }

}

class B extends A {
public void display() {
      System.out.println("I am a method from class B");
   }

}
```

```
class C extends B {
public void display() {
      System.out.println("I am a method from class C");
   }
}
```

Hierarchical inheritance is a type of inheritance in which two or more classes inherit a single parent class. In this, multiple classes acquire properties of the same superclass. The classes that inherit all the attributes or behaviour are known as child classes or subclass or derived classes. The class that is inherited by others is known as a superclass or parent class or base class. For instance, class B, class C, and class D inherit the same class name.

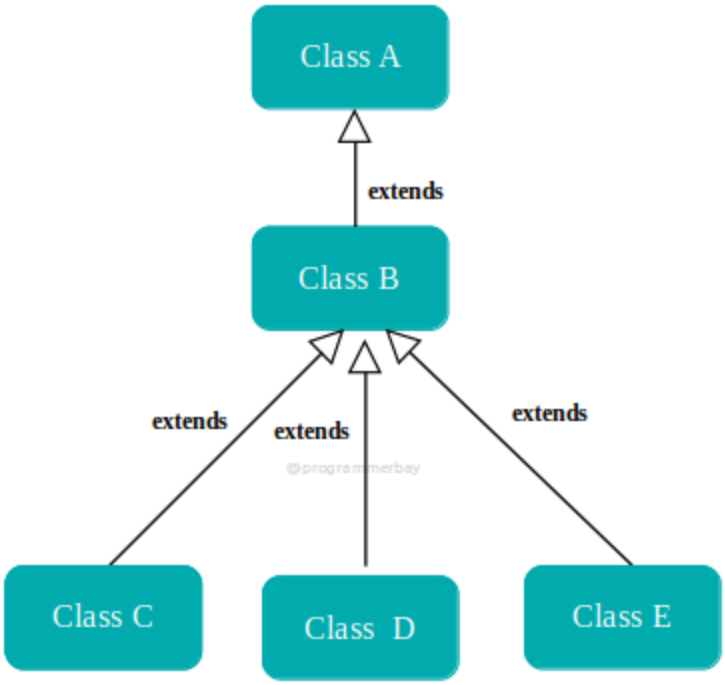



```
public class A {
   public void display() {
      System.out.println("I am a method from class A");
   }
```

```
}

class B extends A {
    public void print() {
        System.out.println("I am a method from class B");
    }
}

class C extends A {
    public void show() {
        System.out.println("I am a method from class C");
    }
}

class D extends A {
    public void outPut() {
        System.out.println("I am a method from class D");
    }

    public static void main(String[] args) {
        B objB = new B();
        C objC = new C();
        D objD = new D();
        objB.display();
        objC.display();
        objD.display();
    }
}
```

Hybrid inheritance is a type of inheritance in which two or more variations of inheritance are used. For instance, suppose, there are various classes namely A, B, C, D and E. if class A gets extended by class B, then this would be an example of Single inheritance. Further, if class C, D and E extend class B,then it

would be an example of hierarchical inheritance. In this manner, multiple types of inheritance can be used within the same hierarchy structure, that is termed Hybrid inheritance.




```
public class A {
   public void display() {
      System.out.println("I am a method from class A");
   }
}

class B extends A {
   public void print() {
      System.out.println("I am a method from class B");
   }
}

class C extends B {
   public void show() {
```

```
        System.out.println("I am a method from class C");
    }
}

class D extends C {
    public void show() {
        System.out.println("I am a method from class D");
    }
    public static void main(String[] args) {
        D objD = new D();
        objD.display(); // A is indirect parent of D, therefore, the display method gets
    }
}
```

Association, Composition and Aggregation in Java

Association in Java

Association, in general terms, refers to the **relationship between any two entities**. Association in java is the relationship that can be established between any two classes. These relationships can be of four types:

1. One-to-One relation
2. One-to-many relation
3. Many-to-one relation
4. Many-to-many relation

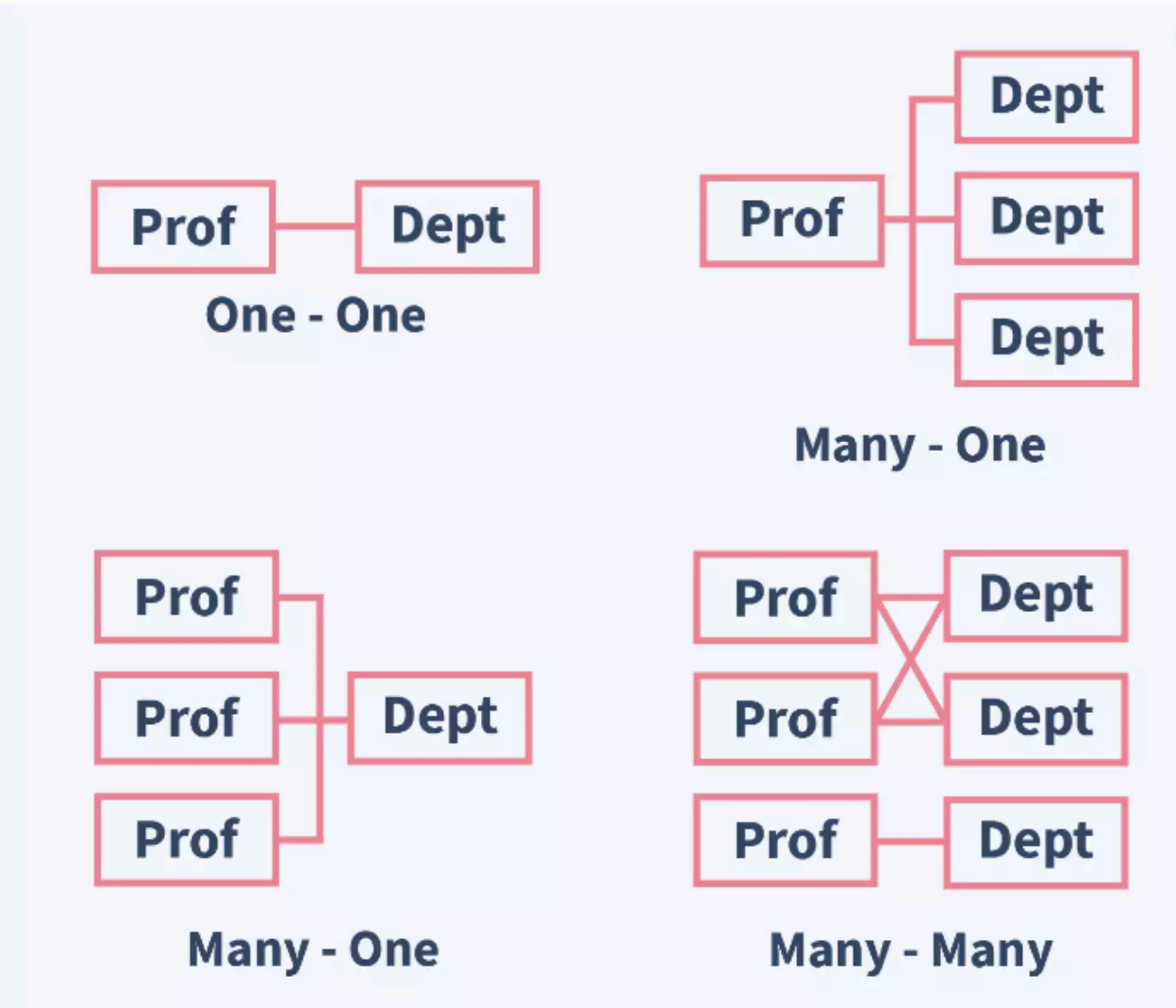
1. **One-to-one association:** A person can have only one Aadhar Card. This defines the one-to-one relationship between the two entities.
2. **One-to-many association:** A teacher explains the concept to many students in a classroom. This defines one-to-many associations between the teacher

and the students.

3. **Many-to-one association:** In an office, many different types of departments are related to an employee. For example, an employee has to report to the financing department for his salary, and he has to report to his manager about the report, the employee is related to the head for any reviews in his project. This is an example of many-to-one associations, where there are many departments related to the employee.
4. **Many-to-many association:** In a cinema hall, the projected movie is meant for a wider audience and not just for one person. There are many actors in the film portraying their messages to many people through the movie. This is an example of many-to-many associations, where many different actors are related to many other people.

Two classes, Professor class, and Department class. Below are the type of relationships/associations that can be possible between them

- 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.



- 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.

```
class Person {
  String name;
  double id;
  //constructor for defining the class objects
  Person(String name, double id2) {
    this.name = name;
    this.id = id2;
  }
class AadharCard extends Person {
  String personName;
  AadharCard(String name, double id) {
    super(name, id);
    this.personName = name;
  }
}
public class AadharBank {
  public static void main(String args[]) {
    AadharCard obj = new AadharCard("PUNEET", 7905470858);
    System.out.println(obj.personName + " is a person with
an Aadhar number: " + obj.id);
  }
}
```

```
import java.util.*;
 class Mobile {
    private String mobile_no;

    public String getMobileNo() {
        return mobile_no;
    }
    public void setMobileNo(String mobile_no) {
        this.mobile_no = mobile_no;
    }
}
```

```
class Person {
   private String name;
   List<Mobile> numbers;
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public List<Mobile> getNumbers() {
      return numbers;
   }
   public void setNumbers(List<Mobile> numbers) {
      this.numbers = numbers;
   }
}
public class AssociationExample {
     public static void main(String[] args) {
          Person person = new Person();
          person.setName("Puneet Vishwakarma");

         Mobile number1 = new Mobile();
          number1.setMobileNo("7905470858");
          Mobile number2 = new Mobile();
          number2.setMobileNo("9044735526");

          List<Mobile> numberList = new ArrayList<Mobile>();
          numberList.add(number1);
          numberList.add(number2);
          person.setNumbers(numberList);
          System.out.println(person.getNumbers()
+" are mobile numbers of the person "+
          person.getName());
       }
```

```
}
```

Forms of Association in Java

- IS-A
- HAS-A

1. IS-A (Inheritance)

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.

2. HAS-A (association)

The HAS-A association on the other hand 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

Ex: A 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.

There are two forms of Association that are possible in Java:

a) Aggregation

b) Composition

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

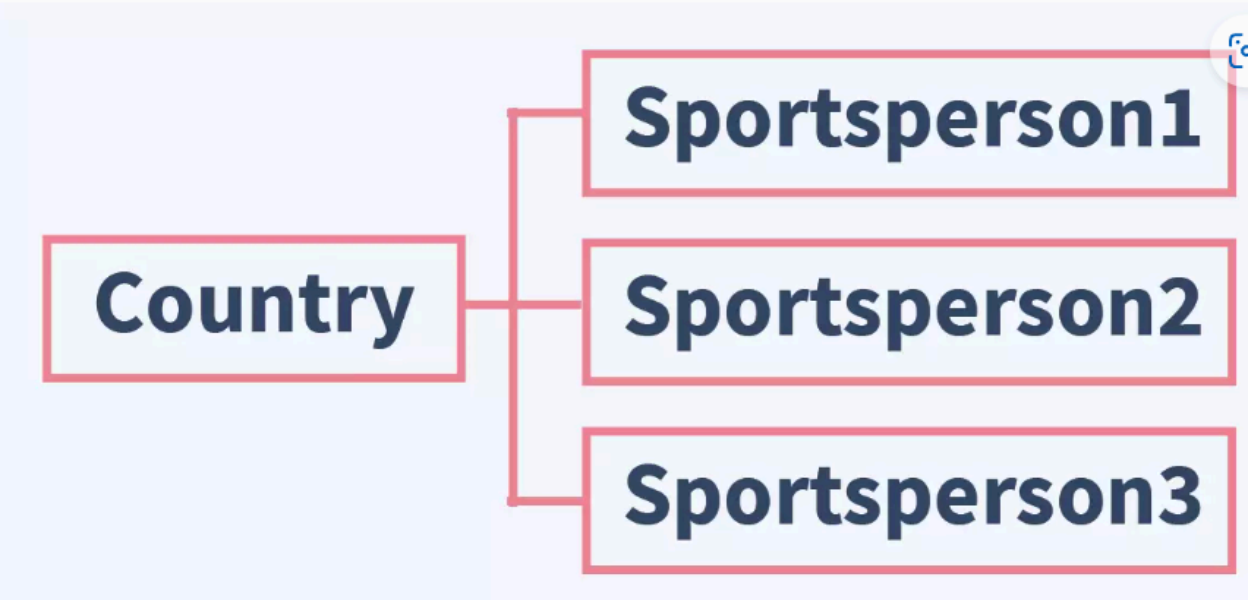
classes are associated with each other, one can exist without the other independently. So Aggregation in java is also called a weak association.

Example: Consider the association between a Country class and a Sportsperson class.

- Country class is defined with a name and other attributes like size, population, capital, etc, and a list of all the Sportspersons that come from it.
- A Sportsperson class is defined with a name and other attributes like age, height, weight, etc.

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.

The Country object has-a Sportsperson object and not the other way around, meaning the Country object instance variables store the Sportsperson objects(This will be clear when we look at the java program in the next section), so the association is one-sided. Thus Aggregation is also known as a unidirectional association.



```
import java.util.*;
// Student class
class Student
{
   String name;
   int enrol ;
   String course;

   Student(String name, int enrol, String course)
   {

      this.name = name;
      this.enrol = enrol;
      this.course = course;
   }
}

// Course class having a list of students.
class course
{

   String name;
   private List<Student> students;

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

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

```
/* College class having a list of Courses*/
class College
{
   String collegeName;
   private List<Course> courses;

   College(String collegeName, List<Course> courses)
   {
      this.collegeName = collegeName;
      this.courses = courses;
   }

   // Returning number of students available in all courses in
//a given college
   public int countStudents()
   {
      int studentsInCollege = 0;
      List<Student> students;

      for(Course course : courses)
      {
         students = course.studentsData();
         for(Student s : students)
         {
            studentsInCollege++;
         }
      }
      return studentsInCollege;
   }
}

// main method
class AggregationExample
{
   public static void main (String[] args)
```

```
  {
     Student std1 = new Student("Emma", 1801, "MCA");
     Student std2 = new Student("Adele", 1802, "BSC-CS");
     Student std3 = new Student("Aria", 1803, "Poly");
     Student std4 = new Student("Ally", 1804, "MCA");
     Student std5 = new Student("Paul", 1805, "Poly");

     // Constructing list of MCA Students.
     List <Student> mca_students = new ArrayList<Student>();
     mca_students.add(std1);
     mca_students.add(std4);

     //Constructing list of BSC-CS Students.
     List <Student> bsc_cs_students = new ArrayList<Student>();
     bsc_cs_students.add(std2);

     //Constructing list of Poly Students.
     List <Student> poly_students = new ArrayList<Student>();
     poly_students.add(std3);
     poly_students.add(std5);

     Course MCA = new Course("MCA", mca_students);
     Course BSC_CS = new Course("BSC-CS", bsc_cs_students);
     Course Poly = new Course("Poly", poly_students);

     List <Course> courses = new ArrayList<Course>();
     courses.add(MCA);
     courses.add(BSC_CS);
     courses.add(Poly);

     // creating object of College.
     College college = new College("ABES", courses);

     System.out.print("Total number of students in the college "+
college.collegeName +" is "+ college.countStudents());
```

```
    }
}
```

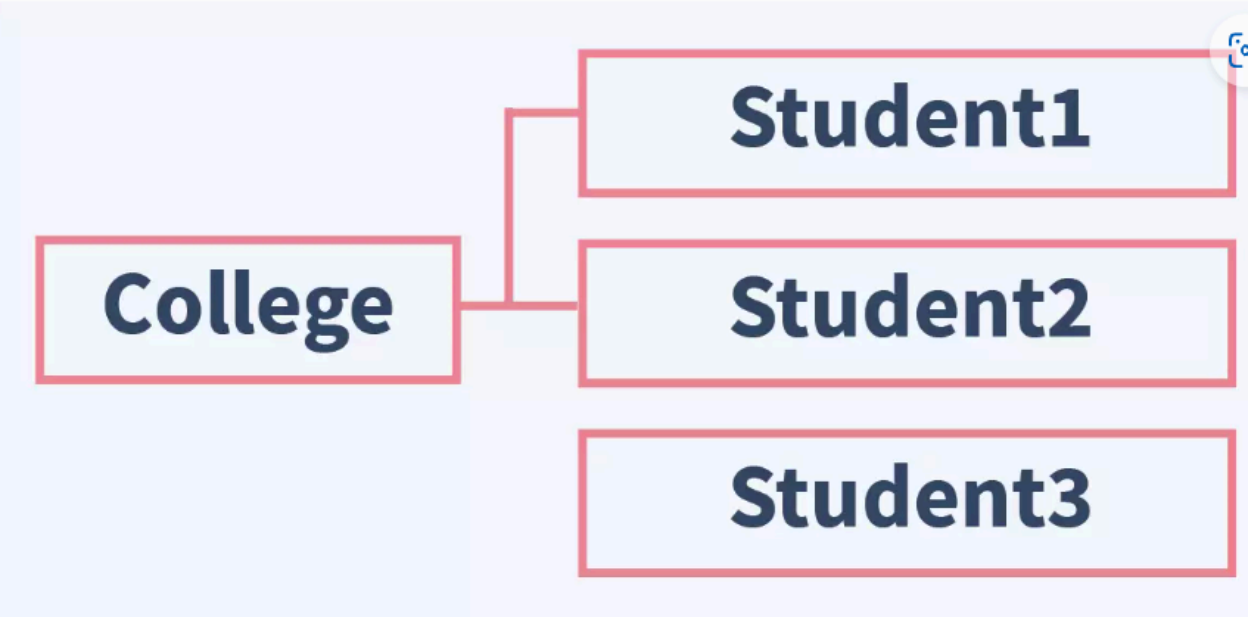
Composition:

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.

Example: The association between College and Student. Below is how it is defined.

- 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.

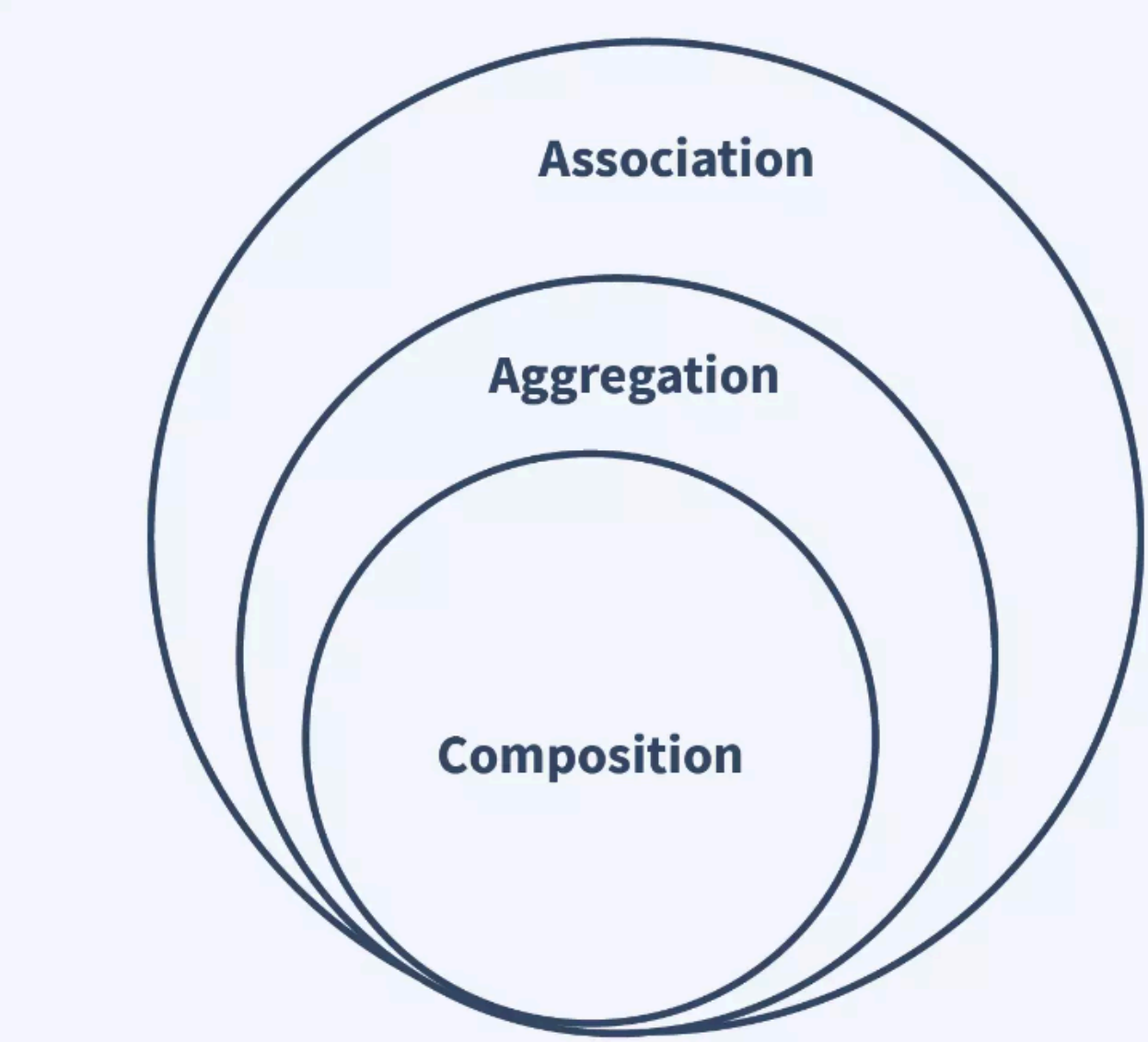


```
import java.util.*;
class Mobile
{
  public String name;
  public String ram;
  public String rom;
  Mobile(String Name, String ram, String rom
  {
    this.name = name;
    this.ram = ram;
    this.rom = rom;
  }
}
class MobileStore
{
  private final List<Mobile> mobiles;
  MobileStore (List<Mobile> mobiles)
  {
    this.mobiles = mobiles;
  }
  public List<Mobile> getTotalMobileInStore(){
    return mobiles;
  }
}
public class CompositionExample {
  public static void main (String[] args)
  {
    Mobile mob1 = new Mobile("Realme6","8GB", "128GB");
    Mobile mob2 = new Mobile("SAMSUNG A21S", "4GB", "128");
    Mobile mob3 = new Mobile("SAMSUNG M10", "4GB", "64GB");
    List<Mobile> mobiles = new ArrayList<Mobile>();
    mobiles.add(mob1);
    mobiles.add(mob2);
    mobiles.add(mob3);
    MobileStore store = new MobileStore(mobiles);
```

```
    List<Mobile> mob = store.getTotalMobileInStore();
    for(Mobile mb : mob){
      System.out.println("Name : " + mb.name + " RAM :" +mb.ram + " and "
          +" ROM : " + mb.rom);
    }
  }
}
```

Difference between association, aggregation, composition in Java

Association in java is one of the types of relations between classes. It has two forms Aggregation(HAS-A) and Composition(Belongs-to). Aggregation is a relatively weak association, whereas Composition is a strong association. Composition can be called a more restricted form of Aggregation. Aggregation can be called the superset of Composition, since all Compositions can are Aggregations but, not all Aggregations can be called Composition.



Problam

Behavior of Instance Variables in case of Inheritance

```
package inheritancePractice;
class P {
   int a = 30;
}
```

```
class Q extends P {
    int a = 50;
}
public class Behavior_of_Instance_Variables extends Q {
    public static void main(String[] args) {
        Q q = new Q();
        System.out.println(" Value of a: " + q.a);
        P p = new Q();
        System.out.println("Value of a: " + p.a);
    }
}
```

Behavior of Overriding method in case of Inheritance

```
package inheritancePractice;

class Baseclass {
    int x = 20;
    // Overridden method.
    void first() {
        System.out.println("Base class method");
    }
}

class Childclass extends Baseclass {
    int x = 50;
    int y = 100;

    // Overriding method.
    void first() {
        System.out.println("Child class first method");
    }

    void second() {
```

```
        System.out.println("Child class second method");
    }
}

public class Behavior_of_Overriding_method {
    public static void main(String[] args) {
        Childclass obj1 = new Childclass();
        System.out.println("Value of x: " + obj1.x); // x of class Child class is called.
        obj1.first(); // msg() of Child class is called.
        obj1.second(); // msg2() of Child class is called.

        Baseclass obj2 = new Childclass();
        System.out.println("Value of x: " + obj2.x); // x of Base class is called.

        // System.out.println("Value of y: " +obj2.y); // Error because y does not exis
        // in Base class.
        obj2.first(); // msg() of Child class is called.
        // obj2.msg2(); // Error because the method msg2() does not exist in Base c
    }
}
```

```
package inheritancePractice;
class Hello {
// Declare an instance block.
    {show();}

    Hello() {
        System.out.println("Hello constructor");
        show();
    }

    void show() {
        System.out.println("Hello method");
    }
}
```

```
class Hi extends Hello {
    Hi() {System.out.println("Hi constructor");}

    void show() { // Override the show() method.
        System.out.println("Hi method");
    }
}

public class Behavior_of_Overriding_method2 {
    public static void main(String[] args) {
        Hi obj = new Hi();
        obj.show(); // show() method of Hi class is called.

        // Superclass reference is equal to child class object.
        Hello obj1 = new Hi();
        obj1.show();
    }
}
```

Behavior of Overloaded method in Inheritance

```
package inheritancePractice;
class Animal {
    void food() {
        System.out.println("What kind of food do lions eat?");
    }
}
class Lion extends Animal {
    void food(int x) {
        System.out.println("Lions eat flesh");
    }
}
public class Behavior_of_Overloaded_method {
```

```
    public static void main(String[] args) {
        Animal a = new Lion();
        a.food(); // food() method of class Animal is called.
        // a.food(20); // Compile time error.

        Lion l = new Lion();
        l.food(); // food() method of class Lion is called.
        l.food(10); // food() method of class Lion is called.
    }
}
```

Key points:

1. At the compile time, an object reference variable plays an important role to call the method.

2. At runtime, the type of object created plays an important role to call the method.

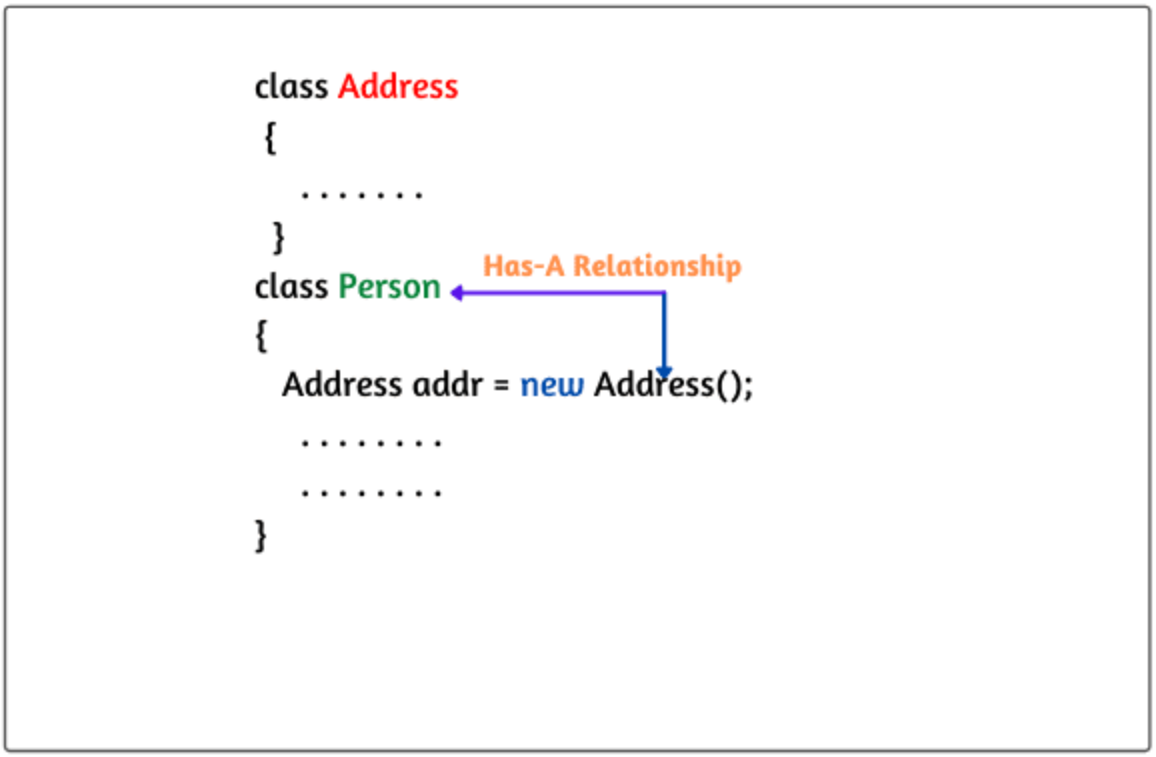
Has-A Relationship

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.

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

```
// Other codes go here.
}
```



Types of Has-A Relationship in Java

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.

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.

Has-A Relationship