PASS BY VALUE AND REFERENCE, RETURN OBJECTS, ASSOCIATION

Pass-by-Value

When a parameter is pass-by-value, the caller and the callee method operate on two different variables which are copies of each other. **Any changes to one variable don't modify the other**.

Pass by value makes a copy in memory of the parameter's value, or a copy of the contents of the parameter. Here is an example in Java:

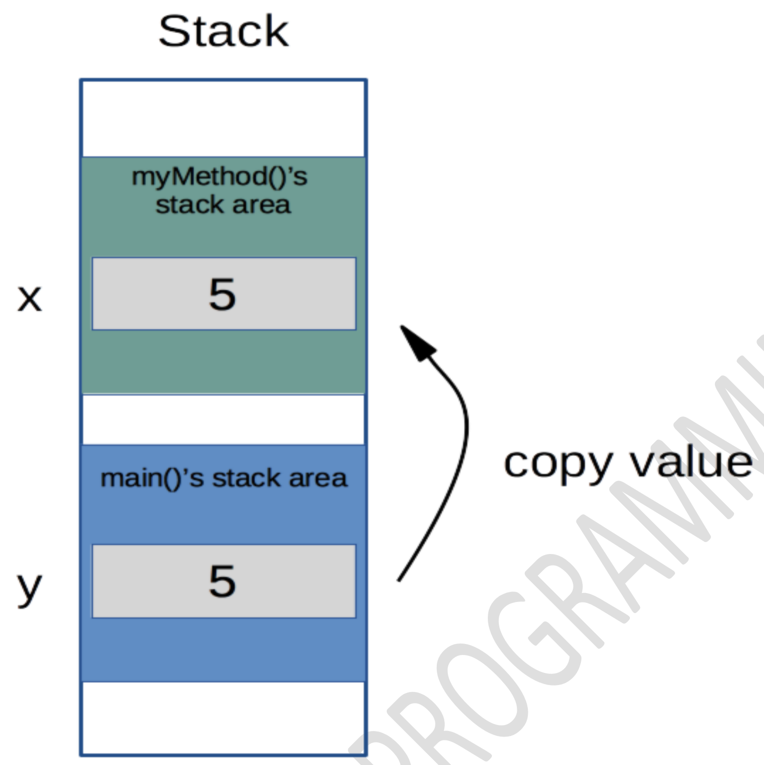
```
public static void main(String[] args) {
   int y = 5;
   System.out.println(y); // prints "5"
   myMethod(y);
   System.out.println(y); // prints "5"
}

public static void myMethod(int x) {
   x = 4; // myMethod has a copy of x, so it doesn't
       // overwrite the value of variable y used
       // in main() that called myMethod
}
```

OUTPUT:

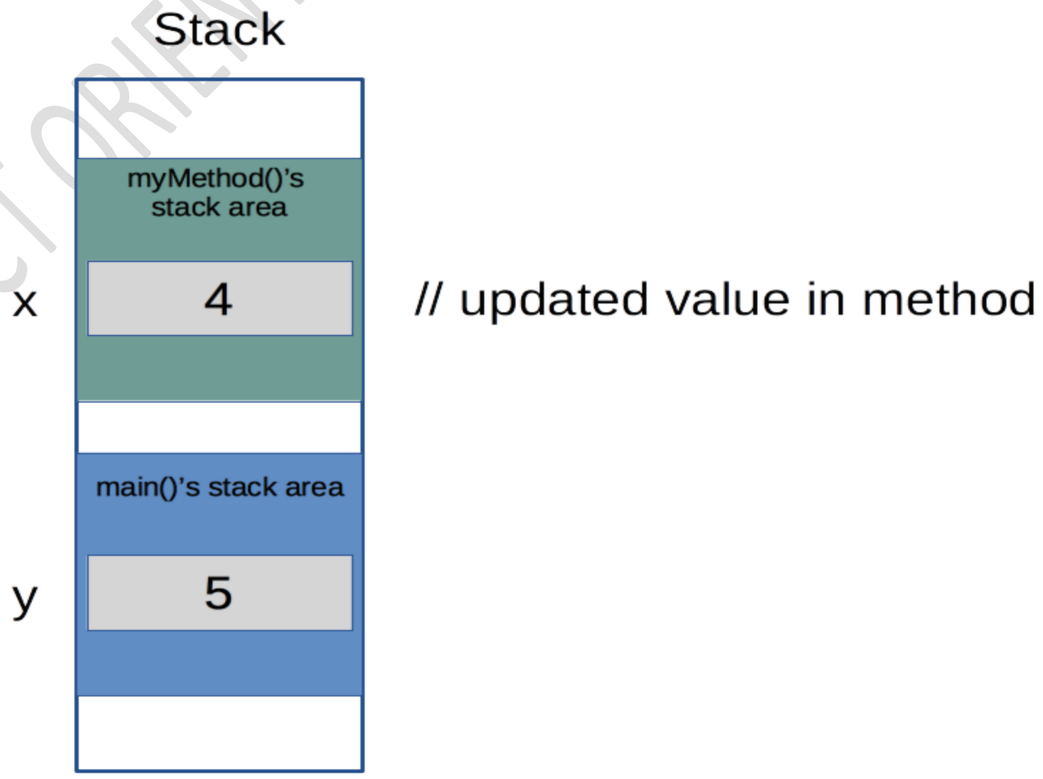
```
5
5
```

In memory, this will look like the following:

Call myMethod which creates a copy of y's value inside variable x on the stack.



myMethod set's x = 4, changing x's value while leaving y's untouched.



```
// another example of pass by value
class Lab6{
   public static void main(String[] args) {
      int a = 10;
      System.out.println(a);


      int b;
      b=a;


      System.out.println(b);


      b=100;
      System.out.println(b);
   }
}
```

OUTPUT

```
10
10
100
```

Pass-by-Reference

When a parameter is pass-by-reference, the caller and the callee operate on the same object.

```
//pass by reference
class Student{
  public String name;
  public int age;
  public float marks;
}

class Lab6{
   public static void main(String[] args) {
      Student abc = new Student();
      abc.name = "Albo Raza";
      abc.age = 29;
      abc.marks = 30.5f;

      System.out.println(abc.name);
      System.out.println(abc.age);
      System.out.println(abc.marks);

      Student xyz;
      xyz = abc;

      System.out.println(xyz.name);
      System.out.println(xyz.age);
      System.out.println(xyz.marks);

      xyz.name = "Ai Bot";
```

```
        xyz.age = 20;
        xyz.marks = 49.5f;

        System.out.println(xyz.name);
        System.out.println(xyz.age);
        System.out.println(xyz.marks);

        System.out.println(abc.name);
        System.out.println(abc.age);
        System.out.println(abc.marks);
    }
}
```

OUTPUT

```
Albo Raza
29
30.5
Albo Raza
29
30.5

Ai Bot
20
49.5
Ai Bot
20
49.5
```

// another example of pass by reference

```
class Example {
        public int a; // making a public member variable
     public Example() {
       a = 10;
    }
    public static void add(Example obj) {
       obj.a++;      // add() method starts here that increments 'a' by 1
    }

    public static void main(String[] args) {
        Example eg = new Example();

       // Before calling the add() method
       System.out.println("Before calling method: " +eg.a);
        // calling method
       add(eg);
        // After calling the add() method
       System.out.println("after calling method: " +eg.a);

       Example ie = eg;
       System.out.println(ie.a);
    }
 }
```

OUTPUT

```
Before calling method: 10
after calling method: 11
11
```

Returning Objects from Methods

Like any other data datatype, a method can returns object. For example, in the following program, the makeTwice( ) method returns an object in which the value of instance variable is two times than it is in the invoking object.

Program (Sample.java)

```
/**
 * This program demonstrates how a method can return
 * a reference to an object.
 */
public class Sample
{
  private int value;

  public Sample(int i)
  {
    value = i;
  }

  /**
   * The makeTwice method returns a Sample object
   * containing the value twice the passed to it.
   */
  public Sample makeTwice()
  {
    Sample temp = new Sample(value * 2);

    return temp;
  }
  public void show()
  {
    System.out.println("Value : " + value);
  }
}

public class ReturnObjectDemo
{
  public static void main(String[] args)
  {
    Sample obj1 = new Sample(10);
    Sample obj2;

    // The makeTwice method returns a reference
    obj2 = obj1.makeTwice();
    obj2.show();
  }
}
```

Output :

```
Value : 20
```

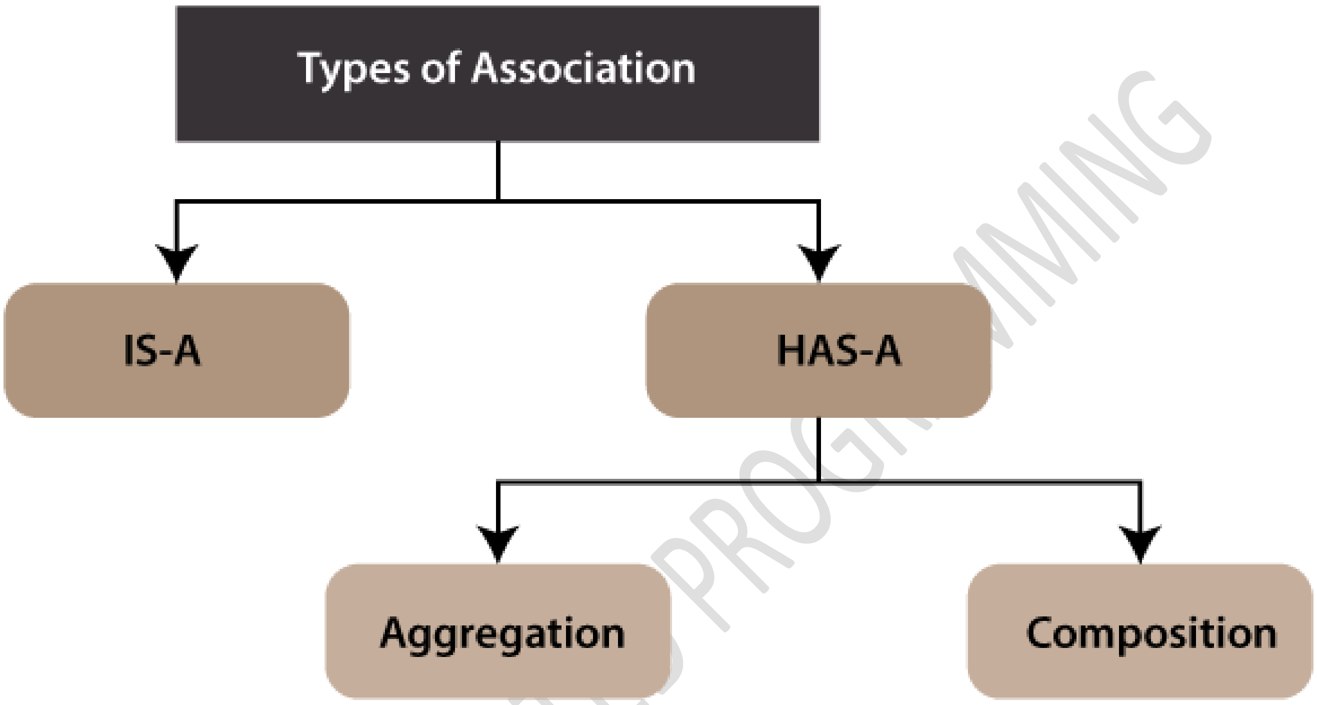
```
// another Example of Returning an object.
class ReturningObject {
   int a;
   ReturningObject(int i) {
      a = i;
   }

   ReturningObject incrByTen() {
      ReturningObject temp = new ReturningObject(this.a + 10);
      return temp;
   }

   public static void main(String args[]) {
      ReturningObject ob1 = new ReturningObject(2);
      ReturningObject ob2;

      ob2 = ob1.incrByTen();
      System.out.println("ob1.a: " + ob1.a);
      System.out.println("ob2.a: " + ob2.a);

      ob2 = ob2.incrByTen();
      System.out.println("ob2.a after second increase: " + ob2.a);
   }
}
```

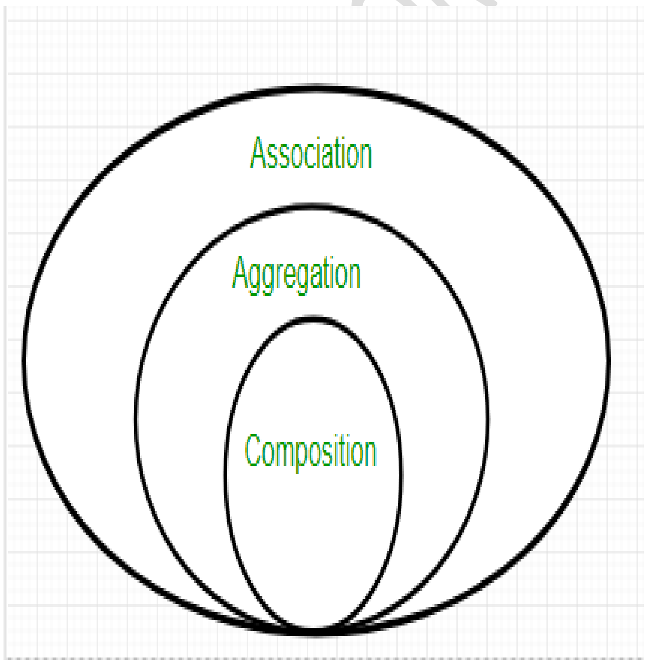
OUTPUT

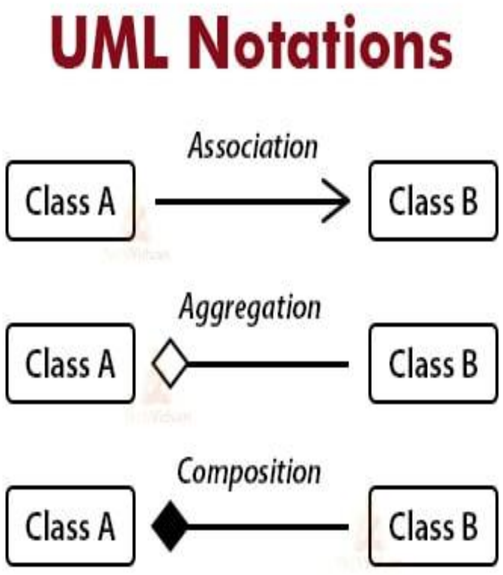
```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

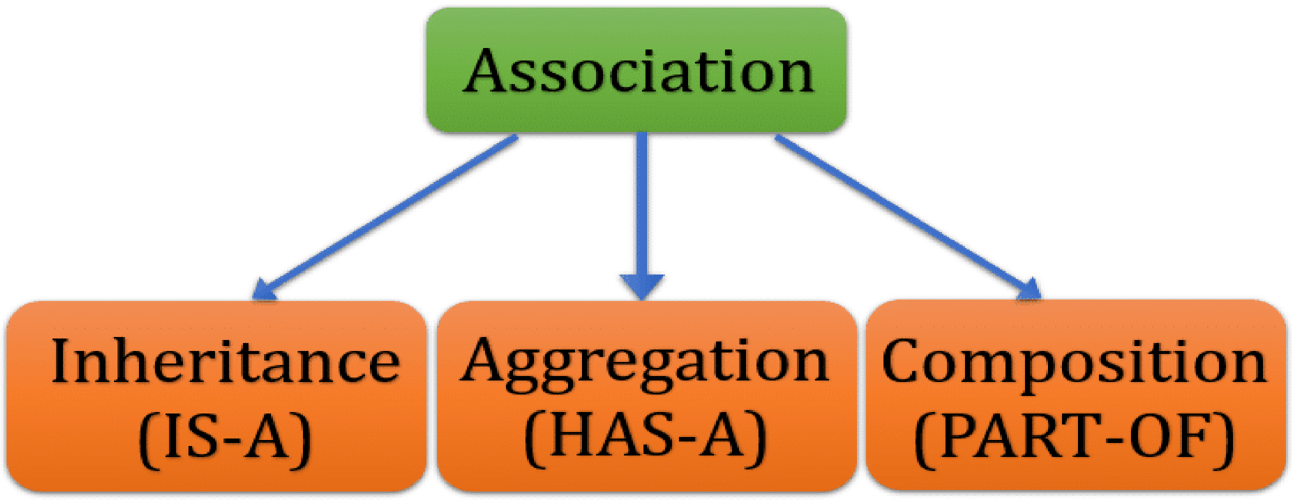
Association in Java

Association in Java is a connection or relation between two separate classes that are set up through their objects. Association relationship indicates how objects know each other and how they are using each other's functionality. In other words In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.









ASSOSIATION (IS-A RELATIONSHIP) INHERITANCE:

An "is-a" relationship in Java represents inheritance, where one class is a subclass of another class. It indicates that an object of a subclass can be treated as an object of its superclass. Here's a simple example:

```
// Superclass
class Animal {
  public void eat() {
    System.out.println("Animal is eating");
  }
}

// Subclass
class Dog extends Animal {
  public void bark() {
    System.out.println("Dog is barking");
  }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog(); // Creating an object of Dog
        dog.eat(); // Inherited method from Animal class
        dog.bark(); // Method specific to Dog class
    }
}
```

In this example:

- The `Animal` class is the superclass, and the `Dog` class is the subclass.
- The `Dog` class "is-a" type of `Animal`, meaning a `Dog` object can be treated as an `Animal`.
- The `Dog` class inherits the `eat()` method from the `Animal` class.
- The `Dog` class also has its own method `bark()`.
- When we create an object of `Dog` and call the `eat()` method, it works as expected because a `Dog` "is-a" type of `Animal`.

This demonstrates the "is-a" relationship where a subclass inherits behavior and attributes from its superclass in Java.

ASSOSIATION (HAS-A RELATIONSHIP) AGGREGATION:

Sure! Here's another simple example demonstrating aggregation:

```
// Class representing a Address
class Address {
    private String street;
    private String city;
    private String state;
    private String zipcode;

    // Constructor
    public Address(String street, String city, String state, String zipcode) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zipcode = zipcode;
    }

    // Method to get full address
    public String getFullAddress() {
        return street + ", " + city + ", " + state + " " + zipcode;
    }
}

// Class representing a Person
class Person {
    private String name;
    private Address address; // Person "has-a" Address
    // Constructor
    public Person(String name, String street, String city, String state, String zipcode) {
        this.name = name;
        this.address = new Address(street, city, state, zipcode); // Initializing Address object when Person is created
    }
    // Method to get full name and address
    public String getFullDetails() {
        return "Name: " + name + ", Address: " + address.getFullAddress();
    }
}
// Main class
public class Main {
    public static void main(String[] args) {
        Person person = new Person("John Doe", "123 Main St", "Springfield", "IL", "12345"); // Creating a Person object
        System.out.println(person.getFullDetails()); // Accessing person's full details
    }
}
```

In this example:

- The `Person` class "has-a" relationship with the `Address` class, as it contains an instance variable `address` of type `Address`.

- In the `Person` constructor, we initialize the `address` object, indicating that every `Person` object will have an associated `Address` object.

- The `Person` class provides a method `getFullDetails()` to access the full name and address of the person it represents.

- The `Address` class represents the address information and can exist independently of the `Person` class.

This demonstrates the "has-a" relationship with aggregation, where the `Person` class contains an `Address` object, but the `Address` object can exist independently of the `Person` class.

ASSOSIATION (HAS-A RELATIONSHIP) COMPOSITION:

A "has-a" relationship in Java represents composition, where one class contains an instance of another class as one of its members. It indicates that an object of one class "has" another object as a part of its state. Here's a simple example:

```
// Class representing a Car
class Car {
    private Engine engine; // Car "has-a" Engine

    // Constructor
    public Car() {
        this.engine = new Engine(); // Initializing Engine object when Car is created
    }

    // Method to start the car
    public void start() {
        engine.start(); // Delegating the start operation to the Engine object
        System.out.println("Car is started");
    }
}

// Class representing an Engine
class Engine {
    // Method to start the engine
    public void start() {
        System.out.println("Engine is started");
    }
}
```

```
// Main class
public class Main {
    public static void main(String[] args) {
        Car car = new Car(); // Creating a Car object
        car.start(); // Starting the car
    }
}
```

In this example:

- The `Car` class "has-a" relationship with the `Engine` class, as it contains an instance variable `engine` of type `Engine`.
- In the `Car` constructor, we initialize the `engine` object, indicating that every `Car` object will have an associated `Engine` object.
- The `Car` class delegates the task of starting the car to the `Engine` object by calling the `start()` method of the `Engine` class from its own `start()` method.

This demonstrates the "has-a" relationship, where the `Car` class contains an `Engine` object as one of its components, and the behavior of the `Car` class is achieved by utilizing the behavior of the `Engine` class.