**Inheritance (IS-A) vs. Composition (HAS-A) Relationship**

Last update on February 26 2020 08:07:31 (UTC/GMT +8 hours)

Description

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the vimplementation of inheritance (IS-A relationship), or object composition (HAS-A relationship). Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition.

**IS-A Relationship:**

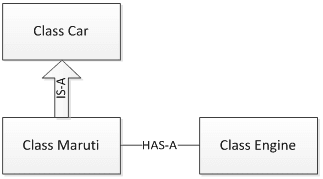
In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

**HAS-A Relationship:**

Composition(HAS-A) simply mean the use of instance variables that are references to other objects. For example Maruti has Engine, or House has Bathroom.

Let’s understand these concepts with an example of Car class.



package relationships;

class Car {

// Methods implementation and class/Instance members

private String color;

private int maxSpeed;

public void carInfo(){

System.out.println("Car Color= "+color + " Max Speed= " + maxSpeed);

}

public void setColor(String color) {

this.color = color;

}

public void setMaxSpeed(int maxSpeed) {

this.maxSpeed = maxSpeed;

}

}

Copy

As shown above, Car class has a couple of instance variable and few methods. Maruti is a specific type of Car which extends Car class means Maruti IS-A Car.

class Maruti extends Car{

//Maruti extends Car and thus inherits all methods from Car (except final and static)

//Maruti can also define all its specific functionality

public void MarutiStartDemo(){

Engine MarutiEngine = new Engine();

MarutiEngine.start();

}

}

Copy

Maruti class uses Engine object’s start() method via composition. We can say that Maruti class HAS-A Engine.

package relationships;

public class Engine {

public void start(){

System.out.println("Engine Started:");

}

public void stop(){

System.out.println("Engine Stopped:");

}

}

Copy

RelationsDemo class is making object of Maruti class and initialized it. Though Maruti class does not have setColor(), setMaxSpeed() and carInfo() methods still we can use it due to IS-A relationship of Maruti class with Car class.

package relationships;

public class RelationsDemo {

public static void main(String[] args) {

Maruti myMaruti = new Maruti();

myMaruti.setColor("RED");

myMaruti.setMaxSpeed(180);

myMaruti.carInfo();

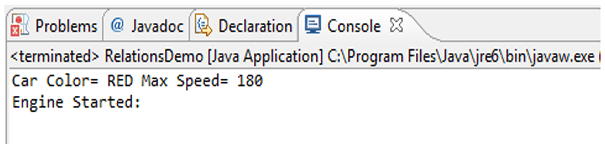
myMaruti.MarutiStartDemo();

}

}

Copy

If we run RelationsDemo class we can see output like below.



**Comparing Composition and Inheritance**

* It is easier to change the class implementing composition than inheritance. The change of a superclass impacts the inheritance hierarchy to subclasses.
* You can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
* Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding)
* It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition) because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.
* With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remains inside the same class.
  + **Don't use inheritance just to get code reuse** If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
  + **Don't use inheritance just to get at polymorphism** If all you really want is a polymorphism, but there is no natural is-a relationship, use composition with interfaces.

**Summary**

* IS-A relationship based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance.
* Has-a relationship is composition relationship which is a productive way of code reuse.