HEAP AREA :

Whenever we create an object in java then the properties and behavior of the object are stored in a special memory area called HEAP AREA.

We have only one HEAP AREA per JVM

STACK AREA :
---

* All the methods whether it is static or non static are executed in a special memory area called Stack Area.

* Whenever we call any method in java then internally one Stack Frame will be created for each and every method.

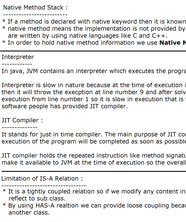
* This Stack Frame contains local and parameter variable, once the method execution is over then stack frame will be deleted.

* JAVA SUPPORTS MULTITHREADING, FOR EACH INDIVIDUAL THREAD WE HAVE A SEPARATE RUNTIME STACK MEMORY. [Multithreading]

* We have multiple Stack memory per JVM.

PC Register :
---

* It stands for Program counter Register.
* Every thread in java contains a separate Stack Memory, In order to hold the **currently executing instruction for each and every thread we have separate PC register as shown below**



Native Method Stack :
---

* If a method is declared with native keyword then it is known as native method.
* native method means the implementation is not provided by using java language, actually these methods are written by using native languages like C and C++.
* In order to hold native method information we use **Native Method Stack.**

Interpreter

In java, JVM contains an interpreter which executes the program line by line.

Interpreter is slow in nature because at the time of execution if we make a mistake at line number 9 then it will throw the exception at line number 9 and after solving the exception again it will start the execution from line number 1 so it is slow in execution that is the reason to boost up the execution java software people has provided JIT compiler.

JIT Compiler :

It stands for just in time compiler. The main purpose of JIT compiler to boost up the execution so the execution of the program will be completed as soon as possible.

JIT compiler holds the repeated instruction like method signature, variables, native method code and make it available to JVM at the time of execution so the overall execution becomes very fast.

Limitation of IS-A Relation :
---

* It is a tightly coupled relation so if we modify any content in the super class then automatically It will reflect to sub class.
* By using HAS-A realtion we can provide loose coupling because Here we can access the property of another class.

What is HAS-A Realtion :
---

* If use another class as a property to current class then it is called HAS-A realtion. By using HAS-A **we can access the property of another class to current class.**

Example :

```
public class Engine
{
  //Engine Properties and behavior
}

public class Car
{
  private Engine engine;  //Car HAS-A engine
}
```

Example 2 :

```
class Order
{
}

class Customer
{
  private Order order; //Customer HAS-A Order
}
```

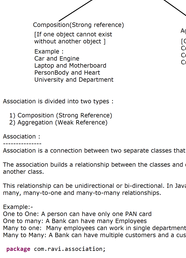
Example 3:

```
class Address
{

}

class Person
{
  private Address address; //Person HAS-A Address
}
```

* HAS-A realtion we can achieve by using Association concept.



Association is divided into two types :

1) Composition (Strong Reference)
2) Aggregation (Weak Reference)

Association :

Association is a connection between two separate classes that can be built up through their Objects.

The association builds a relationship between the classes and describes how much a class knows about another class.

This relationship can be unidirectional or bi-directional. In Java, the association can have one-to-one, one-to-many, many-to-one and many-to-many relationships.

Example:-
One to One: A person can have only one PAN card
One to many: A Bank can have many Employees
Many to one: Many employees can work in single department
Many to Many: A Bank can have multiple customers and a customer can have multiple bank accounts.

```
package com.ravi.association;

public class Student
{
    private int id;
    private String name;
    private double marks;

    public Student(int id, String name, double marks)
    {
        super();
        this.id = id;
        this.name = name;
        this.marks = marks;
    }

    @Override
    public String toString() {
        return "Student [id=" + id + ", name=" + name + ", marks=" + marks + "]";
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getMarks() {
        return marks;
    }

    public void setMarks(double marks) {
        this.marks = marks;
    }



}
```

```
package com.ravi.association;

import java.util.Scanner;

public class Trainer
{
  public static void viewStudentProfile(Student obj) //obj = student
  {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter the Student id :");

      int id = Integer.parseInt(sc.nextLine());

      if(id == obj.getId())
      {
          System.out.println(obj);
      }
      else
      {
          System.err.println("Sorry!!!! id is not available");
      }

  }
}
```

```
package com.ravi.association;

public class AssociationDemo {

    public static void main(String[] args)
    {
        Student student = new Student(101, "Raj", 80);

        Trainer.viewStudentProfile(student);

    }

}
```

Composition :

Composition (Strong reference) :

Composition in Java is a way to design classes such that one class contains an object of another class. It is a way of establishing a "HAS-A" relationship between classes.

Composition represents a strong relationship between the containing class and the contained class.If the containing object (Car object) is destroyed, all the contained objects (Engine object) are also destroyed.

A car has an engine. Composition makes strong relationship between the objects. It means that if we destroy the owner object, its members will be also destroyed with it. For example, if the Car is destroyed the engine will also be destroyed as well.

Program Guidelines :
---

1) One object can't exist without another object
2) We will not create two separate objects, during the creation of Car object, Engine object should be automatically created.
3) We can declare blank final field.

```
package com.ravi.composition;

public class Engine
{
    private String engineType;
    private int horsePower;

    public Engine(String engineType, int horsePower)
    {
        super();
        this.engineType = engineType;
        this.horsePower = horsePower;
    }

    @Override
    public String toString()
    {
        return "Engine [engineType=" + engineType + ", horsePower=" + horsePower + "]";
    }
}
```

```
package com.ravi.composition;

public class Car
{
    private String carName;
    private int carModel;
    private final Engine engine; // HAS-A Relation

    public Car(String carName, int carModel)
    {
        super();
        this.carName = carName;
        this.carModel = carModel;
        this.engine = new Engine("Petrol", 1400);  //composition
        //this.engine = new Engine("CNG", 1250); //not possible blank final field
    }

    @Override
    public String toString()
    {
        return "Car [carName=" + carName + ", carModel=" + carModel + ", engine=" + engine + "]";
    }
}
```

```
package com.ravi.composition;

public class CompositionDemo
{
    public static void main(String[] args)
    {
        Car car = new Car("Wagnor", 2025);
        System.out.println(car);
    }
}
```