Case 2 :
----
* We **cannot** override static method with non static method becaue static belongs to class whereas non static method belongs to object.

* If we try to override static method by using non static method then It will generate CE i.e overridden method is ststic as shown in the below program :

```
class Super
{
     public static void m1()
     {
     }
}

class Sub extends Super
{
     public void m1()
     {
     }
}
public class MethodHidingDemo1
{
     public static void main(String[] args)
     {
          System.out.println("Hello World!");
     }
}
```

Case 3 :
----
* We cannot override non static method with static method, if we try to do the same then It will generate compilation error Overriding method is static.

```
class Super
{
     public void m1()
     {
     }
}
class Sub extends Super
{
     public static void m1()
     {
     }
}
public class MethodHidingDemo1
{
     public static void main(String[] args)
     {
          System.out.println("Hello World!");
     }
}
```

Case 4 :
----
* A static method of super class is bydefault available to sub class so from sub class we can directly call the static method of super class by uisng **class name OR by using Object reference**

```
class Super
{
     public static void m1()
     {
          System.out.println("Super class static method");
     }
}
class Sub extends Super
{

}

public class MethodHidingDemo1
{
     public static void main(String[] args)
     {
          Sub.m1();

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

          Sub s1 = new Sub();
          s1.m1();
     }
}
```

Case 5 :
----
* static methods are not overridden only non static methods are overridden.

* If we write a **static method** in the **sub class** with **same signature** and **compaitable return type** then It is not an overridden method, Actually It is "Method Hiding".

* Here sub class static method will hide super class static method.

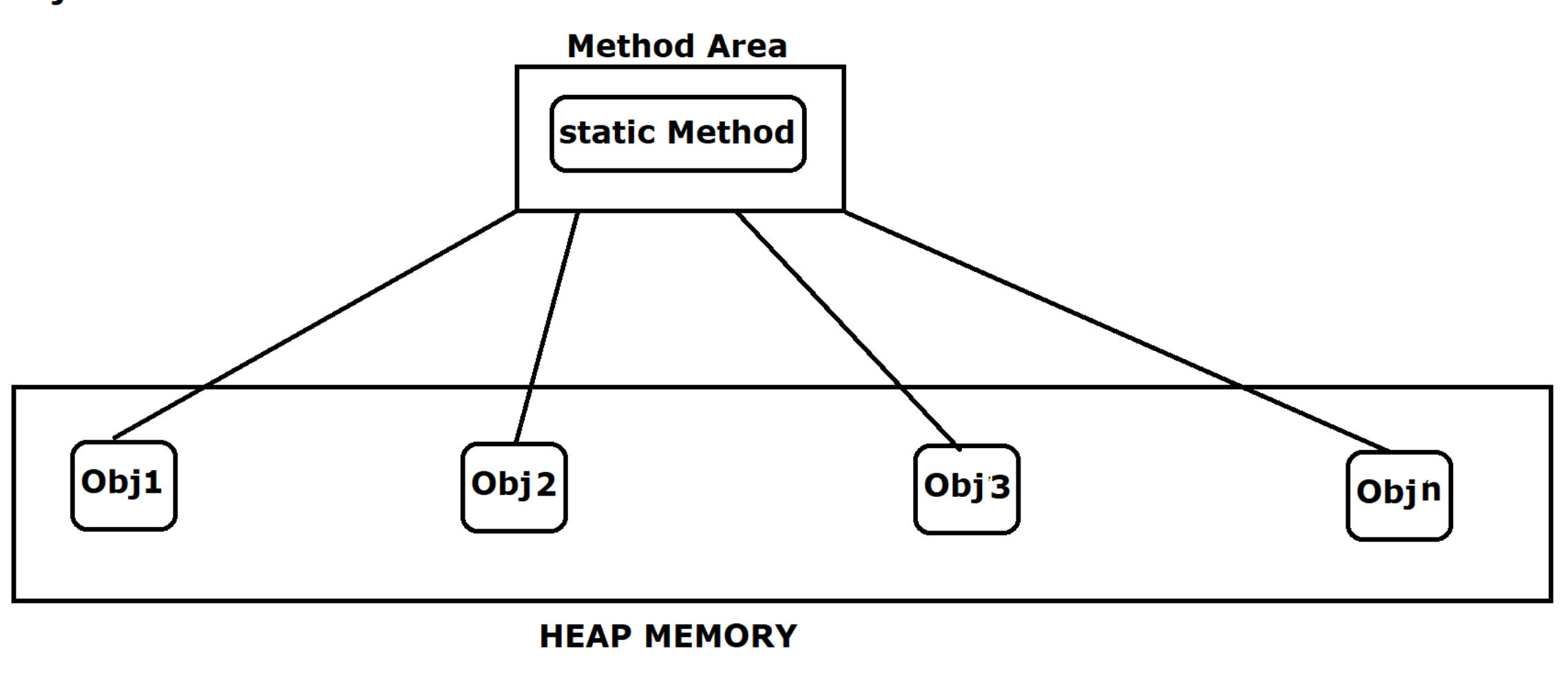
* We cannot use @Override annotation with static method otherwise CE.

```
class Super
{
     public static void m1()
     {
          System.out.println("Super class static method");
     }
}
class Sub extends Super
{

     public static void m1() //Method Hiding
     {
          System.out.println("Sub class static method");
     }
}

public class MethodHidingDemo1
{
     public static void main(String[] args)
     {
       Super s1 = new Sub();
       s1.m1();
     }
}
```

Another Program on Method Hiding

```
package com.ravi.method_hiding;

class Vehicle
{
   public static String name = "Generic Vehicle";

   public static void run()
   {
      System.out.println("Generic Vehicle is Running..");
   }
}

class Bike extends Vehicle
{
    public static String name = "Bike";


   public static void run()
   {
      System.out.println("Bike Vehicle is Running..");
   }
}

class Car extends Vehicle
{
    public static String name = "Car";

   public static void run()
   {
      System.out.println("Car Vehicle is Running..");
   }
}

public class MethodHiding {

   public static void main(String[] args)
   {
      Vehicle v = new Car();
      System.out.println(v.name);
      v.run();
   }

}
```

Note : NSV , SV and static methods are always executed from the current reference regardless of Object.



What is Co-Variant return type :
----
* As we know while overriding a method, Method signature must be same and return type must be compaitable, If we don't have compaitable return type then compiler will generate an error as shown in the below program.

```
class Alpha
{
     public void m1()
     {

     }
}
class Beta extends Alpha
{
     @Override
     public int m1() //error
     {
          return 0;
     }
}
public class CoVariant
{
     public static void main(String[] args)
     {
          System.out.println("Hello World!");
     }
}
```

Note : error, return type int is not compaitable with void.

But from JDK 1.5 onwards we can change the return type of the method in only one case that the return type of both the METHODS(SUPER AND SUB CLASS METHODS) MUST BE IN INHERITANCE RELATIONSHIP (IS-A relationship so it is compatible) called Co-Variant as shown in the program below.

Note :- Co-variant will not work with primitive data type, it will work only with reference type.

***Co-variant represents only one direction that means sub class method return type object we can assign super class method return object i.e in one direction.

```
package com.ravi.co_variant;

class Animal
{}

class Dog extends Animal
{
}

class Cat extends Animal
{

}


class Alpha
{
    public Animal m1()
    {
        System.out.println("Alpha class M1 method");
        return null;
    }
}
class Beta extends Alpha
{
    @Override
    public Dog m1()
    {
        System.out.println("Beta class M1 method");
        return null;
    }
}

public class CoVariantDemo1
{
    public static void main(String[] args)
    {
        Alpha a1 = new Beta();
        a1.m1();

    }

}
```

```
package com.ravi.co_variant;

class Super
{
    public Object m1()
    {
        System.out.println("M1 method of Super class");
        return null;
    }
}
class Sub extends Super
{
    public String m1()
    {
        System.out.println("M1 method of Sub class");
        return null;
    }
}

public class CovariantDemo2
{
    public static void main(String[] args)
    {
        Super s1 = new Sub();
        s1.m1();

    }

}
```

Note : IN ORDER TO UNDERSTAND CO-VARIANT, JUST ASK ONE QUESTION ?
CAN WE ASSIGN SUB CLASS OVERRIDDEN METHOD RETURN TYPE OBJECT TO SUPER CLASS METHOD RETURN TYPE THEN ONLY IT IS CO-VARIANT.

---

final keyword in java :
----
* In order to provide some kind of restriction we should use final keyword.

* final keyword we can use 3 ways in java :

1) To declare a class as a final (Inheritance is not possible)
2) To declare a method as a final (Overriding is not possible)
3) To declare a field/variable as a final (Re-assignemnt is not possible)

To declare a class as a final :
----
* If we declare a class as a final then we cannot extend OR inherit that final class.
* It is used to prevent Inheritance.
* If our class composition/logic of the class is **very important** and we don't want to share to the sub class developer to modify the original behavior of class then we should declare a class as a final.

*** String and all the Wrapper classes are declared as final so inheritance is not possible.**

```
//Programs :
-----------------
final class A
{
     private int x = 100;

     public void setData()
     {
          x = 120;
          System.out.println(x);
     }
}
class B extends A
{
}
public class FinalClassEx
{
     public static void main(String[] args)
     {
          B b1 = new B();
          b1.setData();
     }
}
```

We can create an Object for final class

```
final class Test
{
     private int data;

     public Test(int data)
     {
          this.data = data;
          System.out.println("Data value is :"+data);
     }
}
public class FinalClassEx1
{
     public static void main(String[] args)
     {
          Test t1 = new Test(200);
     }
}
```