

# NULL Object Pattern

[Understanding the need for the NULL Object Pattern](#)

[Code: Return NULL and NULL Checks Included](#)

[NULL Object Design Pattern](#)

[Key Points](#)

[Class Diagram](#)

[Structure of NULL Object Pattern](#)

[Implementation](#)

[Output](#)

[Benefits](#)

## Resources

- [YouTube: Low Level Design from Basics to Advanced \(Some Initial Videos are in Hindi, rest in English\)](#)
- [YouTube: 15. LLD of NULL Object Pattern \(Hindi\) | Design Null Object Pattern | Design Patterns](#)

## Understanding the need for the NULL Object Pattern

1. **Problem:** Code without NULL checks leads to unexpected behaviours due to `NullPointerException`, resulting in a poor user experience. This is also a bad programming practice.

```
private static void printVehicleDetails(Vehicle vehicle){  
  
    System.out.println("Seating Capacity: " + vehicle.getSeatingCapacity());  
    System.out.println("Fuel Tank Capacity: " + vehicle.getTankCapacity());  
}
```

2. **Solution:** A way to solve this problem is to add NULL checks wherever necessary.

```
private static void printVehicleDetails(Vehicle vehicle) {  
    if (vehicle != null) {  
        System.out.println("Seating Capacity: " + vehicle.getSeatingCapacity());  
        System.out.println("Fuel Tank Capacity: " + vehicle.getTankCapacity());  
    }  
}
```

## Code: Return NULL and NULL Checks Included

```
1 public abstract class Vehicle {  
2  
3     public abstract void start();  
4  
5     public abstract void stop();  
6  
7 }  
  
1 public class Car extends Vehicle {  
2     private String model;  
3     private String color;  
4     private int seatingCapacity;  
5     private int fuelTankCapacity;  
6     private boolean isAvailableForTestDrive;  
7  
8     public Car(String model, String color, int seatingCapacity, int  
9         fuelTankCapacity, boolean isAvailableForTestDrive) {  
10         this.model = model;  
11         this.color = color;
```

```

11         this.seatingCapacity = seatingCapacity;
12         this.fuelTankCapacity = fuelTankCapacity;
13         this.isAvailableForTestDrive = isAvailableForTestDrive;
14     }
15
16     @Override
17     public void start() {
18         System.out.println("Car is started and moving");
19     }
20
21     @Override
22     public void stop() {
23         System.out.println("Car is stopped");
24     }
25
26     // Getters
27     public String getModel() {
28         return model;
29     }
30
31     public String getColor() {
32         return color;
33     }
34
35     public int getSeatingCapacity() {
36         return seatingCapacity;
37     }
38
39     public int getFuelTankCapacity() {
40         return fuelTankCapacity;
41     }
42
43     public boolean isAvailableForTestDrive() {
44         return isAvailableForTestDrive;
45     }
46 }
47
48 public class Bike extends Vehicle {
49     private String model;
50     private String color;
51     private int seatingCapacity;
52     private int fuelTankCapacity;
53     private boolean isAvailableForTestDrive;
54
55     public Bike(String model, String color, int fuelTankCapacity) {
56         this.model = model;
57         this.color = color;
58         this.fuelTankCapacity = fuelTankCapacity;
59         this.isAvailableForTestDrive = false;
60         this.seatingCapacity = 2;
61     }
62
63     @Override
64     public void start() {
65         System.out.println("Bike is started and moving");
66     }
67
68     @Override
69     public void stop() {
70         System.out.println("Bike is stopped");
71     }
72
73     // Getters
74     public String getModel() {
75         return model;
76     }
77
78     public String getColor() {
79         return color;
80     }
81

```

```

82     public int getSeatingCapacity() {
83         return seatingCapacity;
84     }
85
86     public int getFuelTankCapacity() {
87         return fuelTankCapacity;
88     }
89
90     public boolean isAvailableForTestDrive() {
91         return isAvailableForTestDrive;
92     }
93 }

```

```

1  public class VehicleFactory {
2
3      public static Vehicle getVehicle(String type) {
4          if (type.equals("car")) {
5              return new Car("Toyota", "Red", 5, 60, true);
6          } else if (type.equals("bike")) {
7              return new Bike("Yamaha", "Black", 60);
8          } else {
9              return null; // THE PROBLEM
10             }
11         }
12     }
13
14     public class ProblemDemo {
15         public static void main(String[] args) {
16             System.out.println("##### Null Object Pattern: Problem Demo
17             #####");
18
19             Vehicle car = VehicleFactory.getVehicle("car");
20             printVehicleDetails(car);
21             testDrive(car);
22
23             Vehicle bike = VehicleFactory.getVehicle("bike");
24             printVehicleDetails(bike);
25             testDrive(car);
26
27             // Saved by NULL Check in printVehicleDetails and testDrive
28             methods
29             // Without NULL Check, it will not throw NullPointerException
30             or ClassCastException
31             Vehicle nullVehicle = VehicleFactory.getVehicle("null");
32             printVehicleDetails(nullVehicle);
33             testDrive(nullVehicle);
34         }
35
36         private static void printVehicleDetails(Vehicle vehicle) {
37             if (vehicle != null) { // THE PROBLEM
38                 if (vehicle instanceof Car car) {
39                     System.out.print("\n[+] Vehicle Details: ");
40                     System.out.println(car.getClass().getSimpleName() + "
41                     [Model=" + car.getModel()
42                     + ", Color=" + car.getColor() + ", Seating
43                     Capacity=" + car.getSeatingCapacity()
44                     + ", Fuel Tank Capacity=" +
45                     car.getFuelTankCapacity() + "]);
46                 }
47                 if (vehicle instanceof Bike bike) { // THE PROBLEM
48                     System.out.print("\n[+] Vehicle Details: ");
49                     System.out.println(bike.getClass().getSimpleName() + "
50                     [Model=" + bike.getModel()
51                     + ", Color=" + bike.getColor() + ", Fuel Tank
52                     Capacity=" + bike.getFuelTankCapacity() + "]);
53                 }
54             }
55
56         private static void testDrive(Vehicle vehicle) {
57             if (vehicle != null) { // THE PROBLEM

```

```

51         vehicle.start();
52         vehicle.stop();
53     }
54 }
55 }

```

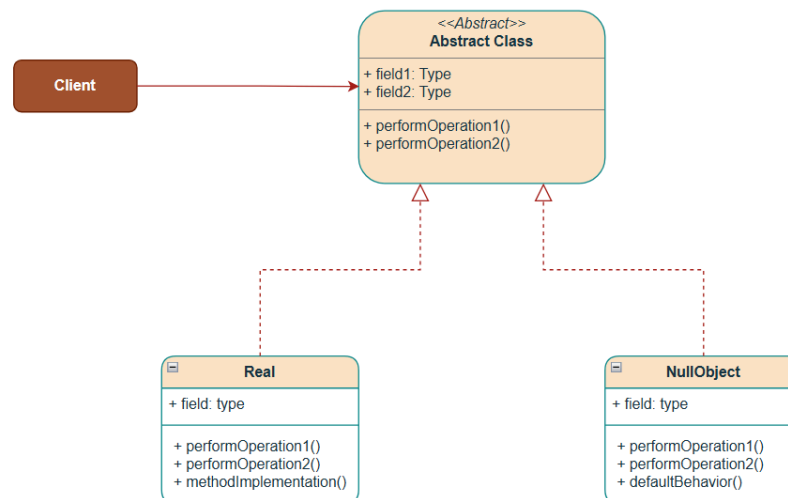
## NULL Object Design Pattern

The NULL Object Pattern is a behavioral design pattern that **uses polymorphism to eliminate null checks**. Instead of **returning NULL** and **adding NULL checks** wherever necessary, we return a special object called **NULL OBJECT** that implements the expected interface( or extends an abstract class) but does nothing (or provides default behavior).

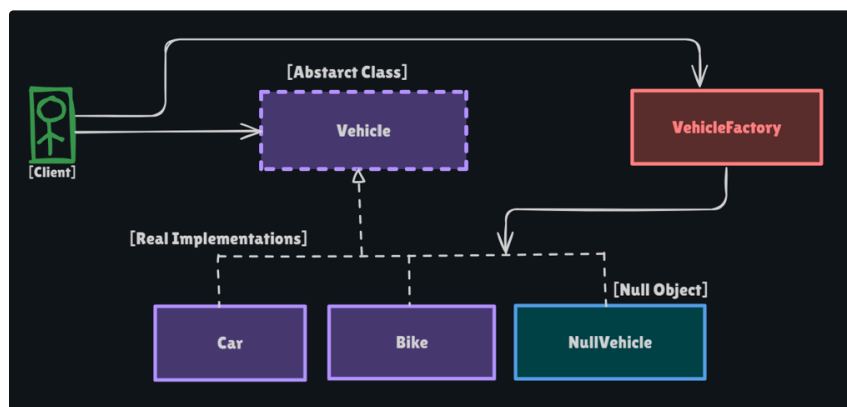
### Key Points

- Instead of returning null, return an instance of a Null Object. A NULL Object replaces a NULL return type.
- This leads to clean code without redundant NULL checks everywhere.
- NULL Object reflects do Nothing or contains a Default behaviour.

### Class Diagram



### Structure of NULL Object Pattern



1. **Interface:** `Vehicle` defines the contract to be implemented by `Real` & `NullObject` classes provide respective behaviour.
2. **Real Implementations:** `Car` and `Bike` do the actual work.
3. **NULL Object:** `NullVehicle` implements the interface but does nothing. Provides default properties & behaviour.
4. **Factory:** `VehicleFactory` receives `vehicle-type` as client input to create and return a `Real` specific implementation of the `Vehicle` object. If the client provides a non-existent vehicle type, the factory returns a `NullVehicle` instance.

## Implementation

The `Vehicle`, `Car`, and `Bike` classes remain unchanged as we add the `NullVehicle` class to the hierarchy, allowing the Factory to return a `NullObject` with default behaviour when the requested vehicle type doesn't match.

```
1 public class NullVehicle extends Vehicle {
2     private final String model;
3     private final String color;
4     private final int seatingCapacity;
5     private final int fuelTankCapacity;
6     private final boolean isAvailableForTestDrive;
7
8     public NullVehicle() {
9         this.model = "Default";
10        this.color = "Default";
11        this.seatingCapacity = 0;
12        this.fuelTankCapacity = 0;
13        this.isAvailableForTestDrive = false;
14    }
15
16    @Override
17    public void start() {
18        // Do nothing - silent Vehicle
19        System.out.print("\n[-] Null Vehicle: start() - do nothing");
20    }
21
22    @Override
23    public void stop() {
24        // Do nothing - silent Vehicle
25        System.out.println("\n[-] Null Vehicle: stop() - do nothing");
26    }
27
28    // Getters
29    public int getSeatingCapacity() {
30        return seatingCapacity;
31    }
32
33    public int getFuelTankCapacity() {
34        return fuelTankCapacity;
35    }
36
37    public boolean isAvailableForTestDrive() {
38        return isAvailableForTestDrive;
39    }
40 }
```

```
1 public class VehicleFactory {
2
3     public static Vehicle getVehicle(String type) {
4         if (type.equals("car")) {
5             return new Car("Toyota", "Red", 5, 60, true);
```

```

6         } else if (type.equals("bike")) {
7             return new Bike("Yamaha", "Black", 60);
8         } else {
9             return new NullVehicle(); // THE SOLUTION
10        }
11    }
12 }

```

```

1 public class SolutionDemo {
2     public static void main(String[] args) {
3         System.out.println("\n#### Null Object Pattern: Solution Demo
4         #####");
5
6         Vehicle car = VehicleFactory.getVehicle("car");
7         printVehicleDetails(car);
8         testDrive(car);
9
10        Vehicle bike = VehicleFactory.getVehicle("bike");
11        printVehicleDetails(bike);
12        testDrive(car);
13
14        // Saved by NULL Check in printVehicleDetails and testDrive
15        methods
16        // Without NULL Check, it will not throw NullPointerException
17        or ClassCastException
18        Vehicle nullVehicle = VehicleFactory.getVehicle("null");
19        printVehicleDetails(nullVehicle);
20        testDrive(nullVehicle);
21    }
22
23    private static void printVehicleDetails(Vehicle vehicle) {
24        if (vehicle instanceof Car car) {
25            System.out.print("\n[+] Vehicle Details: ");
26            System.out.println(car.getClass().getSimpleName() + "
27            [Model=" + car.getModel()
28            + ", Color=" + car.getColor() + ", Seating
29            Capacity=" + car.getSeatingCapacity()
30            + ", Fuel Tank Capacity=" +
31            car.getFuelTankCapacity() + "]\n");
32        }
33        if (vehicle instanceof Bike bike) {
34            System.out.print("\n[+] Vehicle Details: ");
35            System.out.println(bike.getClass().getSimpleName() + "
36            [Model=" + bike.getModel()
37            + ", Color=" + bike.getColor() + ", Fuel Tank
38            Capacity=" + bike.getFuelTankCapacity() + "]\n");
39        }
40    }
41
42    private static void testDrive(Vehicle vehicle) {
43        vehicle.start();
44        vehicle.stop();
45    }
46 }

```

```

private static void printVehicleDetails(Vehicle vehicle) {
    if (vehicle instanceof Car car) {
        System.out.print("\n[+] Vehicle Details: ");
        System.out.println(car.getClass().getSimpleName() + " [Model=" + car.getModel()
        + ", Color=" + car.getColor() + ", Seating Capacity=" + car.getSeatingCapacity()
        + ", Fuel Tank Capacity=" + car.getFuelTankCapacity() + "]\n");
    }
    if (vehicle instanceof Bike bike) {
        System.out.print("\n[+] Vehicle Details: ");
        System.out.println(bike.getClass().getSimpleName() + " [Model=" + bike.getModel()
        + ", Color=" + bike.getColor() + ", Fuel Tank Capacity=" + bike.getFuelTankCapacity() + "]\n");
    }
}

```

This is a Java pattern-matching feature introduced in Java 16.

This line does two things at once:

1. It checks if the `vehicle` object is an instance of the `Car` class.
2. If it is, it automatically casts the `vehicle` to the `Car` type and assigns it to a new variable named `car` that can be used within the if block.

## Output

```
##### Null Object Pattern: Solution Demo #####

[+] Vehicle Details: Car [Model=Toyota, Color=Red, Seating Capacity=5, Fuel Tank Capacity=60]
Car is started and moving
Car is stopped

[+] Vehicle Details: Bike [Model=Yamaha, Color=Black, Fuel Tank Capacity=60]
Car is started and moving
Car is stopped

[-] Null Vehicle: start() - do nothing
[-] Null Vehicle: stop() - do nothing

Process finished with exit code 0
```

## Benefits

- Makes code cleaner by eliminating repetitive null checks.
- Reduces `NullPointerException` risks.
- Increases code readability.