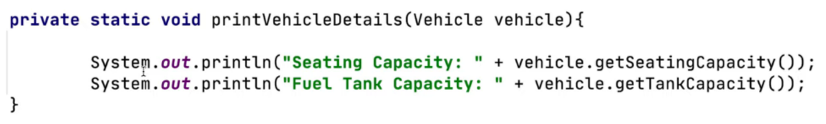
NULL Object Pattern

Resources

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

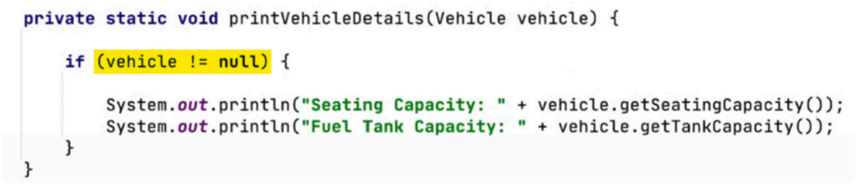
Understanding the need for the NULL Object Pattern

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



```java
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.



```java
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

```java
public abstract class Vehicle {

    public abstract void start();

    public abstract void stop();

}
```

```java
public class Car extends Vehicle {
  private String model;
    private String color;
    private int seatingCapacity;
    private int fuelTankCapacity;
    private boolean isAvailableForTestDrive;

    public Car(String model, String color, int seatingCapacity, int fuelTankCapacity, boolean isAvailableForTestDrive) {
        this.model = model;
        this.color = color;
        this.seatingCapacity = seatingCapacity;
        this.fuelTankCapacity = fuelTankCapacity;
        this.isAvailableForTestDrive = isAvailableForTestDrive;
    }

    @Override
    public void start() {
        System.out.println("Car is started and moving");
    }

    @Override
    public void stop() {
        System.out.println("Car is stopped");
    }

    // Getters
    public String getModel() {
      return model;
    }

    public String getColor() {
        return color;
    }

    public int getSeatingCapacity() {
        return seatingCapacity;
    }

    public int getFuelTankCapacity() {
        return fuelTankCapacity;
    }

    public boolean isAvailableForTestDrive() {
        return isAvailableForTestDrive;
    }
}

public class Bike extends Vehicle {
    private String model;
    private String color;
    private int seatingCapacity;
    private int fuelTankCapacity;
    private boolean isAvailableForTestDrive;

    public Bike(String model, String color, int fuelTankCapacity) {
        this.model = model;
        this.color = color;
        this.fuelTankCapacity = fuelTankCapacity;
        this.isAvailableForTestDrive = false;
        this.seatingCapacity = 2;
    }

    @Override
    public void start() {
        System.out.println("Bike is started and moving");
    }

    @Override
    public void stop() {
        System.out.println("Bike is stopped");
    }

    // Getters
    public String getModel() {
        return model;
    }

    public String getColor() {
        return color;
    }

    public int getSeatingCapacity() {
        return seatingCapacity;
    }

    public int getFuelTankCapacity() {
        return fuelTankCapacity;
    }

    public boolean isAvailableForTestDrive() {
        return isAvailableForTestDrive;
    }
}
```

```java
public class VehicleFactory {

    public static Vehicle getVehicle(String type) {
        if (type.equals("car")) {
            return new Car("Toyota", "Red", 5, 60, true);
        } else if (type.equals("bike")) {
            return new Bike("Yamaha", "Black", 60);
        } else {
            return null; // THE PROBLEM
        }
    }
}
public class ProblemDemo {
    public static void main(String[] args) {
        System.out.println("##### Null Object Pattern: Problem Demo
#####");

        Vehicle car = VehicleFactory.getVehicle("car");
        printVehicleDetails(car);
        testDrive(car);

        Vehicle bike = VehicleFactory.getVehicle("bike");
        printVehicleDetails(bike);
        testDrive(car);

        // Saved by NULL Check in printVehicleDetails and testDrive
methods
        // Without NULL Check, it will not throw NullPointerException
or ClassCastException
        Vehicle nullVehicle = VehicleFactory.getVehicle("null");
        printVehicleDetails(nullVehicle);
        testDrive(nullVehicle);
    }

    private static void printVehicleDetails(Vehicle vehicle) {
        if (vehicle != null) { // THE PROBLEM
            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() + "]");

            }
            if (vehicle instanceof Bike bike) { // THE PROBLEM
                System.out.print("\n[+] Vehicle Details: ");
                System.out.println(bike.getClass().getSimpleName() + "
[Model=" + bike.getModel()
                        + ", Color=" + bike.getColor() + ", Fuel Tank
Capacity=" + bike.getFuelTankCapacity() + "]");
            }
        }
    }

    private static void testDrive(Vehicle vehicle) {
        if (vehicle != null) { // THE PROBLEM
```

```
                vehicle.start();
                vehicle.stop();
            }
        }
}
```

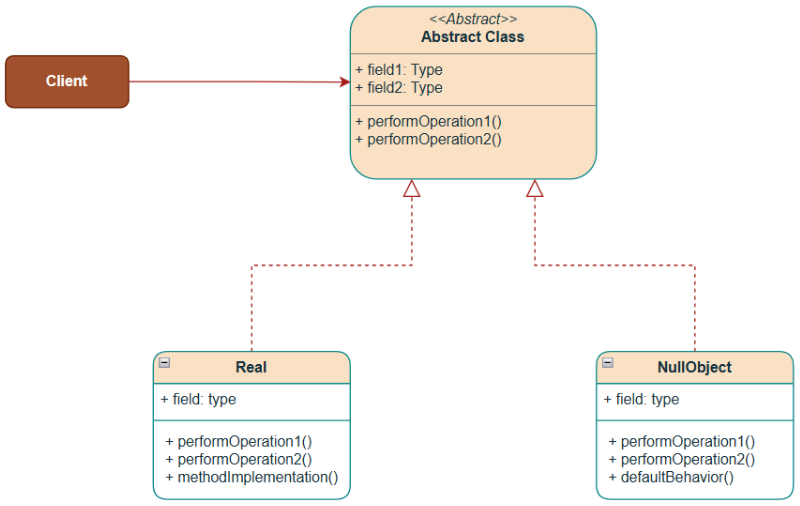
NULL Object Design Pattern

> **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).**

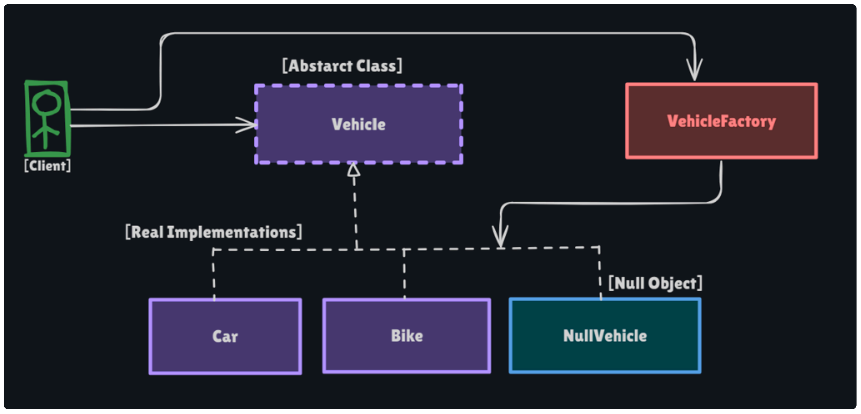
Key Points

- 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.

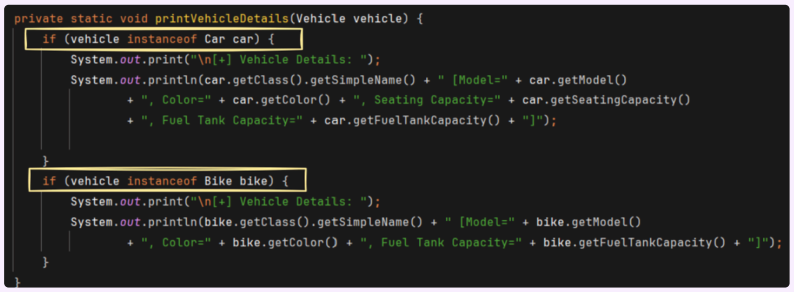
Implementation

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.

```java
public class NullVehicle extends Vehicle {
    private final String model;
    private final String color;
    private final int seatingCapacity;
    private final int fuelTankCapacity;
    private final boolean isAvailableForTestDrive;

    public NullVehicle() {
        this.model = "Default";
        this.color = "Default";
        this.seatingCapacity = 0;
        this.fuelTankCapacity = 0;
        this.isAvailableForTestDrive = false;
    }

    @Override
    public void start() {
        // Do nothing - silent Vehicle
        System.out.print("\n[-] Null Vehicle: start() - do nothing");
    }

    @Override
    public void stop() {
        // Do nothing - silent Vehicle
        System.out.println("\n[-] Null Vehicle: stop() - do nothing");
    }

    // Getters
    public int getSeatingCapacity() {
        return seatingCapacity;
    }

    public int getFuelTankCapacity() {
        return fuelTankCapacity;
    }

    public boolean isAvailableForTestDrive() {
        return isAvailableForTestDrive;
    }
}
```

```java
public class VehicleFactory {

    public static Vehicle getVehicle(String type) {
        if (type.equals("car")) {
            return new Car("Toyota", "Red", 5, 60, true);
        } else if (type.equals("bike")) {
            return new Bike("Yamaha", "Black", 60);
        } else {
            return new NullVehicle(); // THE SOLUTION
        }
    }
}
```

```java
public class SolutionDemo {
    public static void main(String[] args) {
        System.out.println("\n##### Null Object Pattern: Solution Demo #####");

        Vehicle car = VehicleFactory.getVehicle("car");
        printVehicleDetails(car);
        testDrive(car);

        Vehicle bike = VehicleFactory.getVehicle("bike");
        printVehicleDetails(bike);
        testDrive(car);

        // Saved by NULL Check in printVehicleDetails and testDrive methods
        // Without NULL Check, it will not throw NullPointerException or ClassCastException
        Vehicle nullVehicle = VehicleFactory.getVehicle("null");
        printVehicleDetails(nullVehicle);
        testDrive(nullVehicle);
    }

    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() + "]");

            }
            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() + "]");
            }
    }

    private static void testDrive(Vehicle vehicle) {
            vehicle.start();
            vehicle.stop();
    }
}
```



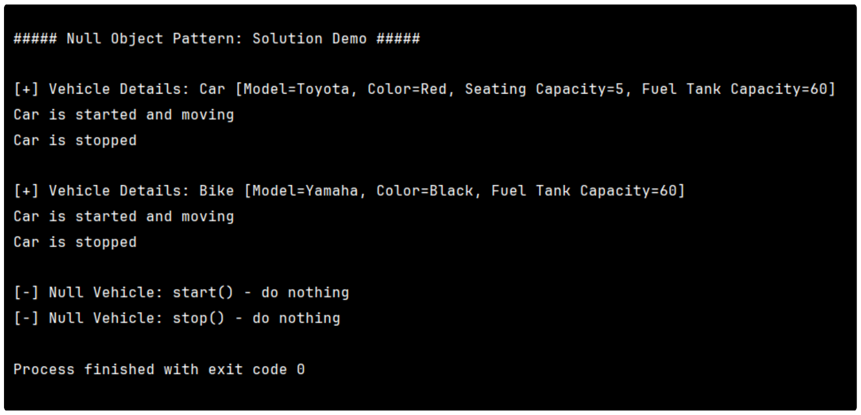
```java
private static void printVehicleDetails(Vehicle vehicle) {
    if (vehicle instanceof Car car) {
        System.out.print("\n[+] Vehicle Details: ");
        System.out.println(car.getClass().getSimpleName() + " [Model=" + car.getModel()
                + ", Color=" + car.getColor() + ", Seating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

    }
    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() + "]");
    }
}
```

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.