

Null Object Pattern

Michal Moravik, SD20w2

Name: Null Object Pattern

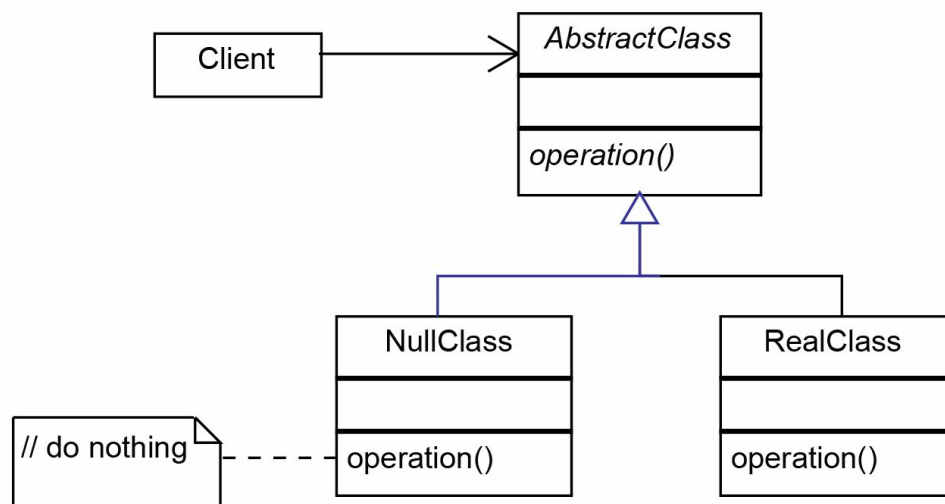
Category: Behavioral pattern

Intent: Provides an empty object (null object) whose functions do nothing. We want to use this object when we want to avoid null pointer exceptions and never-ending checks if some variable is not null. We will create an abstraction (abstract class) for a class we want to instantiate. From this class, we will create the null object class and the real class. Both of which will implement the same functions.

Motivation: Null value can introduce lots of runtime errors to the system. In order to avoid these errors, we would need to constantly check if a variable that should contain an object does not contain the null value instead. In order to avoid these checks and runtime errors, we implement the Null Object Pattern.

The result, the null object, will implement the same functions as the normal object but without any implementation, usually, just a blank function body.

UML:



Implementation:

Declare interface for the objects (real object & null object)

```
public interface IWeapon {  
    void use();  
}
```

Make a null object which implements the interface but does nothing

```
public class NullWeapon implements IWeapon {  
    @Override  
    public void use() {  
        System.out.println("I do nothing man!");  
    }  
}
```

Make a real implementation/s

```
public class AK47Weapon implements IWeapon {  
    @Override  
    public void use() {  
        System.out.println("RATAIA");  
    }  
}
```

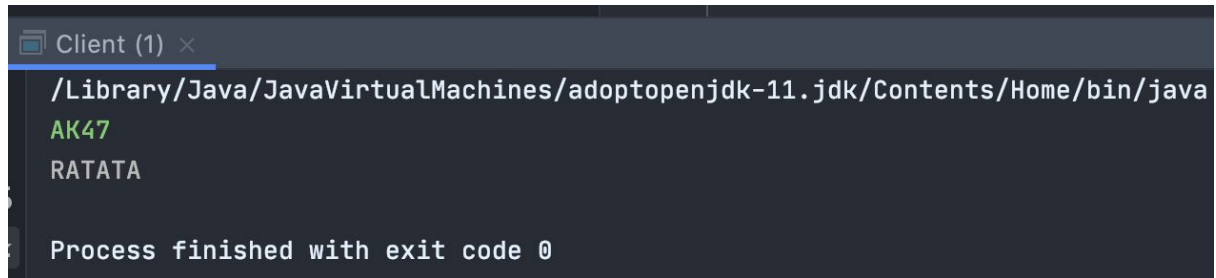
Implement a client to see the advantage of using the null object.

```
public class Client {  
    public static void main(String[] args) throws IOException {  
        // input from user - can vary during the runtime  
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));  
        String chosenWeapon = reader.readLine();  
  
        IWeapon weapon;  
  
        switch (chosenWeapon) {  
            case "AK47":  
                weapon = new AK47Weapon();  
                break;  
            default:  
                weapon = new NullWeapon();  
                break;  
        }  
  
        weapon.use();  
    }  
}
```

If we write anything else than "AK47", we will create an instance of the null object. We can then call the method of that object which should do nothing. We can see it in the output:

```
Client (1) x  
/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home/bin/java  
dummytext  
I do nothing man!  
  
Process finished with exit code 0
```

If we write "AK47" as an input, it will create an AK47Weapon instance and print out "RATATA" which belongs to this concrete class AK47Weapon.



```
Client (1) x
/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home/bin/java
AK47
RATATA

Process finished with exit code 0
```

Consequences:

Pros:

- Will handle null pointer exception runtime error. Thanks to the null object, this method will be called instead and the system won't break.
- We are not forced to check if an object is equal to null in every place and create various solutions on how to deal with nulls. All we need is just a null object.

Cons:

- Can waste memory if there are too many null objects
- Can create a complex system because of implementing interface / abstract class with null class

Known uses:

- Whenever we need to securely handle a possible "null" value. Instead of doing endless checks, we can implement a null object which will ensure that the system is running seamlessly.

Related patterns:

- Singleton pattern - a null object can be implemented as a singleton to save memory usage. But this introduces other cons that are described in my Singleton Pattern ID document.